An Implementable Parallel Scheduler for Input-Queued Switches

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

Abstract—**The input-queued (IQ) switch architecture has received much attention in the research community and with implementors because it scales well with the line speed and the switch size. The main reason for this is that the memory bandwidth requirement for an input-queued switch is minimal, making it less expensive to implement compared to an output-queued or a shared-memory switch. To get a good delay and throughput performance from an IQ switch requires the use of efficient packet scheduling algorithms for matching input and output ports. For example, the maximum weight matching (MWM) algorithm is known to deliver a throughput of upto 100% and to provide low delays. But MWM is complex to implement at high line rates and scales poorly with the switch size. Many algorithms which approximate the performance of MWM have been proposed; but, they are still too complex to implement and/or do not provide a good performance compared to MWM.**

This paper proposes an innovative algorithm, called APSARA, which aims to bridge the gap between good performance and ease of implementation. The main idea is to use limited parallelism to find a matching in a single iteration, as compared to the $O(N^3)$ iterations needed by MWM in the worst case for an $N \times N$ switch. We prove that APSARA achieves a throughput of upto 100% and extensive simulations show that its delay performance is nearly as good as that of MWM.

I. Introduction and Motivation

The high demand for bandwidth on the Internet has seen the introduction of higher and higher speed links, and has caused an attendant demand for routers with a high aggregate switching capacity. At the very highest speeds input-queued switches have become the architecture of choice mainly because the memory bandwidth of their packet buffers is very low compared to that of the output-queued and shared-memory architectures.

In order to perform well, however, 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. The maximum weight matching (MWM) algorithm finds, from amongst the $N!$ possible matchings, that matching whose weight is the highest. Here, the weight of the edge connecting input i to output j can either be the number of packets queued at input i for output j or the age of oldest packet at input i for output j. The MWM is known to provide a throughput of 100% [1], [2], [3] so long as no input or output is over subscribed, and achieves a low average delay by keeping queue-lengths small. However, it is complex to implement – it needs $O(N^3)$ iterations in the worst-case and does not lend to an easy pipelined implementation.

Implementation considerations have therefore seen the proposal of a number of scheduling algorithms for high-speed switches; for example, iSLIP [4], iLQF [5], RPA [6] and MUCS [7]. 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%. Thus, although the above-mentioned algorithms are aimed at the implementation issue, their performance is poor.

This raises the question: Is it possible for an algorithm to compete with the performance of MWM and yet be simple to implement? If yes, what feature of the problem remains to be exploited?

The answer lies in recognizing two features of the high-speed switch scheduling problem: (1) Packets arrive (depart) at most one per input (output) per time slot. This means queue-lengths, which are taken to be the weights by MWM, change very little during successive time slots. This suggests that a heavy matching will continue to be heavy for a few more time slots. (2) Two (randomly chosen) matchings that differ by very few (e.g. two) edges will quite likely be just as heavy. That is, given a heavy matching M, there is quite likely a matching M' that is a "neighbor" of M that is also heavy. This provides a basis for efficiently searching the set of matchings over successive time slots by looking at the neighbors of the current matching.

The above observations when made precise yield the algorithm APSARA which uses parallelism in hardware to search for a good matching in each time slot. More impor-

0-7695-1357-3/01 $10.00 © 2001 IEEE

tantly, it needs *only one* iteration per time slot, regardless of the size of the switch.

The rest of the paper describes the algorithm APSARA, states some interesting theoretical properties, and discusses its implementation.

II. A MODEL OF THE SWITCH

We consider an $N \times N$ input-queued switch. The buffer at input i is partitioned into N "virtual output queues" (VOQs), where VOQ_{ij} stores packets at input i for output j. Following common practice we assume that packets are of fixed length[1] and denote the size of VOQ_{ij} at time t by $q_{ij}(t)$. Let $Q(t) = [q_{ij}(t)]$ be an $N \times N$ matrix capturing the lengths of all VOQs at time t.

Let λ_{ij} be the average rate at which packets arrive at input i for output j, and let $\Lambda = [\lambda_{ij}]$ be the average arrival rate matrix, also called the "load matrix". We require the load matrix to be admissible: $\sum_j \lambda_{ij} \leq 1$ for every i, and $\sum_i \lambda_{ij} \leq 1$ for every j. In words, this condition ensures that no input or output is over subscribed.

The switching fabric is assumed to be internally non-blocking (e.g. a crossbar). Such a fabric places a constraint on scheduling algorithms: In each time slot, each input can connect with at most one output and each output can connect with at most one input. We use the binary variables $x_{ij}(t)$, $i, j = 1, \dots, N$ to denote connections. Input i is connected with output j at time t if, and only if, $x_{ij}(t) = 1$. Without loss of generality, we consider only complete connections; that is, we allow a connection between input i and an output j even if $q_{ij}(t) = 0$. The crossbar constraint can now be modeled in the following way:

$$x_{ij}(t) \in \{0, 1\}, \quad \forall i, j = 1, \dots, N$$
$$\sum_{j=1}^{N} x_{ij}(t) = 1, \quad \forall i = 1, \dots, N$$
$$\sum_{i=1}^{N} x_{ij}(t) = 1, \quad \forall j = 1, \dots, N$$

A feasible connection configuration can be seen as a matching in a bipartite graph, where inputs and outputs correspond to nodes in the graph and an edge between input i and output j denotes that they are connected or matched. Let $X(t) = [x_{ij}(t)]$ denote the matching matrix at time t. Note that for an $N \times N$ switch the set of all possible matchings, denoted by S_N, has a cardinality of $N!$.

It is the job of the switch scheduling algorithm to determine, at each time t, the particular matching that will be used. Thus, for example, the algorithm could decide to connect inputs and outputs in the following round-robin fashion. At time 0, input i connects with output i; at time 1, input i connects with output $(i+1) \mod N$; etc. We can denote the corresponding matching matrices as: $x_{ij}(t) = 1$ if $j = (i+t) \mod N$, for every j, $1 \leq j \leq N$.

A scheduling algorithm of particular interest to us is the maximum weight matching (MWM) algorithm, which we now proceed to define. Denote the "weight" of a matching $X(t) = [x_{ij}(t)]$ as $W(t) = \sum_{i,j} q_{ij}(t) x_{ij}(t)$, taking the weight of the edge between input i and output j to be equal to the queue-length $q_{ij}(t)$. The maximum weight matching algorithm chooses, at each time t, that matching whose weight is the highest. More precisely, if $X^w(t)$ denotes the matching determined by MWM at time t, then $X^w(t)$ is given by

$$X^w(t) = \arg \max_{X \in S_N} \Big\{ \sum_{i,j} x_{ij} q_{ij}(t) \Big\}. \quad (1)$$

It has been shown that for all admissible Bernoulli i.i.d. input traffic patterns MWM delivers upto 100% throughput [1], [2]. The restriction of Bernoulli i.i.d. inputs was later relaxed in [3]. Further, extensive simulations show that it provides low delays. However, the main drawback of MWM is that it is difficult to implement in very high-speed and/or in large-sized switches. This motivates the algorithm we propose in the next section.

III. APSARA

Motivating discussion: As mentioned in the introduction, there are two features of the switch scheduling problem we wish to take advantage of to come up with an easy-to-implement high-performance scheduling algorithm. The following discussion recalls these features and shows, intuitively, why they will help obtain good matchings.

1. The queue-lengths $q_{ij}(\cdot)$ do not change by much between iterations. Indeed, each $q_{ij}(\cdot)$ can increase at most by one due to a possible arrival and decrease at most by one due to a possible departure. This implies that the weight of a matching changes by a bounded amount, making it likely that a heavy matching will tend to stay heavy over several time slots.
2. Two matchings X and Y that differ in very few edges will be called "neighbors". Denote by $\mathcal{N}(X)$ the set of all

[1] Although Internet packets are variable-length it is common for high-speed routers to fragment them into fixed length cells before switching and to reassemble the cells at the egress port into packets.

neighbors of the matching X. The observation we shall exploit is that if X is a heavy matching, there is a very good chance that it has a neighbor $X' \in \mathcal{N}(X)$ which is also heavy. If the cardinality of $\mathcal{N}(X)$ is *small* then this search can be conducted in parallel in hardware.

We will shortly define what we mean by a neighbor and make the above notions precise. For now, we note that the two features work in our favor in the following way: Given the matching, $X(t)$, at time t, we explore $\mathcal{N}(X(t))$ in parallel to determine if there is a matching X' which is heavier than $X(t)$. If yes, we use the heaviest such matching at time $t+1$. Else, we continue to use the matching $X(t)$ at time $t+1$. The two observations made above suggest that the weight of the matching at time $t+1$ is likely to be quite good. The details follow.

Given a matching $X = [x_{ij}]$, let $\pi(i) = j$ if $x_{ij} = 1$. That is, the matching X connects input i to output $\pi(i)$. This allows us to shorten the representation of a matching; for example, suppose $N = 3$ and consider the matching:

$$X = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

It can be represented as the vector $(\pi(1), \pi(2), \pi(3)) = (2, 1, 3)$.

Definition 1. *(Neighbor) A matching Y is said to be a neighbor of the matching X iff there are exactly two inputs, say i_1 and i_2, such that Y 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 X and Y.*

Note that X and Y differ in only two edges, the other $N-2$ edges are the same for both.

Definition 2. *(Neighborhood set) The set of all neighbors of a matching X will be denoted by $\mathcal{N}(X)$. Note that the cardinality of $\mathcal{N}(X)$ is $\binom{N}{2}$.*

As an example, consider a 3×3 switch. The matching X and its 3 neighbors X_1, X_2 and X_3 are given below:

$$X = (1, 2, 3)$$
$$X_1 = (2, 1, 3), \; X_2 = (1, 3, 2), \; X_3 = (3, 2, 1)$$

A. A Hamiltonian walk on matchings

Before presenting the algorithm we need one last concept, that of a Hamiltonian walk on the set of all matchings. We introduce the walk and use it in the description of the APSARA algorithm only because this allows us to prove that APSARA achieves upto 100% throughput. In the section on simulations we do not use this concept, and yet we shall find that APSARA achieves 100% throughput.

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, it visits each of the $N!$ distinct node one after the other exactly once between time $t = 1, \ldots, N!$. 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 [8]. This algorithm produces $Z(t)$ such that, for all t, $Z(t+1)$ is neighbor of $Z(t)$. When this algorithm is executed for $N = 3$ it generates the matchings: $Z(1) = (1,2,3)$, $Z(2) = (1,3,2)$, $Z(3) = (3,1,2)$, $Z(4) = (3,2,1)$, $Z(5) = (2,3,1)$, $Z(6) = (2,1,3)$, and $Z(7) = Z(1)$, $Z(8) = Z(2), \ldots$

B. APSARA: The basic algorithm

Let $X(t)$ be the matching determined by APSARA at time t and let $Q(t+1)$ be the queue-lengths at the beginning of time $t+1$. At time $t+1$ APSARA does the following:

(i) Determine the neighbors, $\mathcal{N}(X(t))$, of $X(t)$ and the matching $Z(t+1)$ corresponding to the Hamiltonian walk at time $t+1$.

(ii) Let $\mathcal{S}(t+1) = \mathcal{N}(X(t)) \cup Z(t+1) \cup X(t)$. Compute the weight of every matching $Y \in \mathcal{S}(t+1)$ as follows:

$$W(Y) = \sum_{ij} y_{ij} q_{ij}(t+1).$$

(iii) The matching at time $t+1$ is given by

$$X(t+1) = \arg \max_{U \in \mathcal{S}(t+1)} \{W(U)\},$$

.

The basic version of the APSARA algorithm described above requires the computation of the weight of neighbor matchings. Note that each such computation is easy since a neighbor Y differs from the matching $X(t)$ in exactly two edges. However, computing the weights of all $\binom{N}{2}$ neighbors, if done in parallel as shown in figure 1, will require a lot of space in hardware for large values of N.

But high-aggregate bandwidth switches come in two flavors: (i) a small number of ports connected to very high-speed lines, or (ii) a large number of ports connected to lower-speed lines. So, if the goal is to build a high-aggregate bandwidth switch with a small number of ports (say, 30-40 ports), then one requires less than 800 modules for computing the weights of neighbor matchings. The big

win in this case is time (APSARA requires only iteration), and for switches connected to high speed lines the time available for scheduling packets is very small. Thus, APSARA helps in this case by trading off space for time.

If, on the other hand, one wants to build a switch with 1000 ports, say, then one needs upto 500,000 modules. This can be prohibitively expensive. We approach this issue from a different direction. Say that hardware space constraints allow the use of at most $K \ll N^2$ modules, then how can the search procedure required by APSARA 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.

C. APSARA: A randomized variant

Suppose hardware constraints allow the use of only K modules. Given the matching used at time t, $X(t)$, and the queue-lengths $Q(t+1)$, the matching $X(t+1)$ is determined as follows:

(i) Pick K elements uniformly at random from the set $\mathcal{N}(X(t))$. Let $\mathcal{N}_K(X(t))$ denote the set of these elements. Note that it is not necessary to generate $\mathcal{N}(X(t))$. Determine the matching $Z(t+1)$ corresponding to the Hamiltonian walk at time $t+1$.

(ii) Let $\mathcal{S}_K(t+1) = \mathcal{N}_K(X(t)) \cup Z(t+1) \cup X(t)$. For every $Y \in \mathcal{S}_K(t+1)$, compute

$$W(Y) = \sum_{ij} y_{ij} q_{ij}(t+1).$$

(iii) Then

$$X(t+1) = \arg \max_{U \in \mathcal{S}_K(t+1)} \{W(U)\},$$

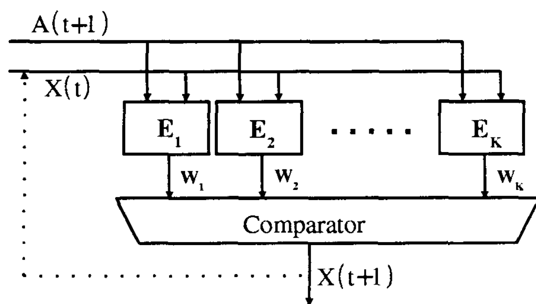

Fig. 1. A schematic for the implementation of APSARA. The old matching, $X(t)$, and the new arrivals, $A(t+1)$, are used to compute the weights of neighbor matchings in parallel. The new matching, $X(t+1)$, is determined as a result of weight comparison.

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 $X(t)$ 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 $X(t)$ determined by APSARA into a maximal matching. We shall call the maximal version Max-APSARA.

D. APSARA: Throughput Theorem

We state the following theorem, whose proof is not provided in this extended abstract due to space limitations.

Theorem 1. *APSARA is stable (i.e. achieves upto 100% throughput) for any admissible Bernoulli i.i.d. packet arrival process.*

IV. Performance

We compare the performance of APSARA with that of other known algorithms: iSLIP and iLQF (both run upto N iterations) and MWM. As gentle reminder, we do not use the matching $Z(t)$, given by the Hamiltonian walk process. While this version of APSARA should perform worse (since it has one less matching at its disposal), we shall see that even this version performs quite well, giving upto 100% throughput and good delays.

A. Simulation settings

Switch: Switch size: $N = 32$. Each VOQ can store upto 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 is the most commonly used test traffic in the literature.

2. Diagonal: $\lambda_{ii} = 2\rho/3N$, $\lambda_{i|i+1|} = \rho/3N \ \forall i$, 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.

Performance measures: We compare performance of different algorithms by measuring the mean input queue-lengths. The delays can be computed directly using Little's formula.

We let the simulation run until the estimate of the average delay reaches the relative width of confidence interval equal to 1% with probability ≥ 0.95. The estimation of the confidence interval width is obtained using the *batch means* approach.

B. Simulation results: Basic version

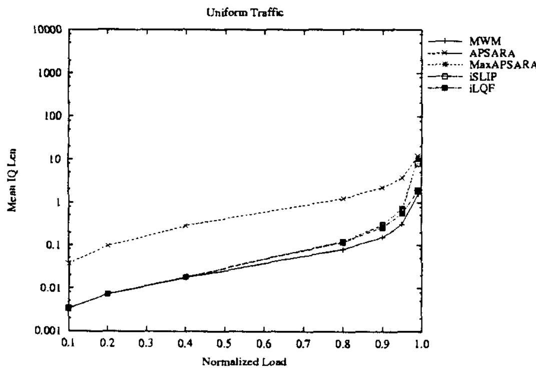

Fig. 2. Mean IQ length for uniform traffic. APSARA is well-behaved but its queue occupancies are greater than all the other algorithms for low loads, since it is not maximal.

Figures 2, 3 and 4 show the mean input queue-lengths for uniform, diagonal and logdiagonal loading. Among all the algorithms considered APSARA is the only *non-maximal* algorithm, in the sense that for low load some additional connections could be added in the schedule. This explains why APSARA shows delays greater than all the other algorithms for low load. Max-APSARA is able to bridge the gap in the delays, and the effect of the maximizing the matching decreases with the increasing load, implying that APSARA becomes maximal at high loads. The main observation from these simulation results is that both APSARA and Max-APSARA are able to reach 100% throughput under all possible traffic loading.

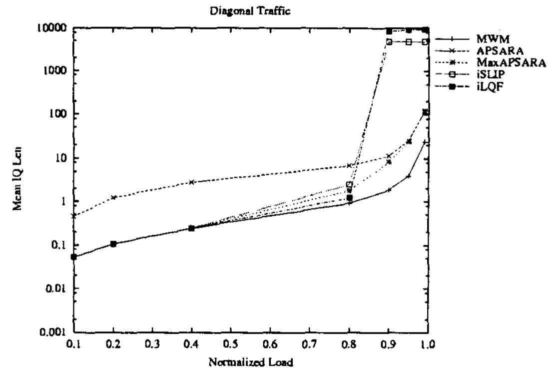

Fig. 3. Mean IQ length for diagonal traffic. APSARA is the only algorithm able to approximate the MWM with bounded delays.

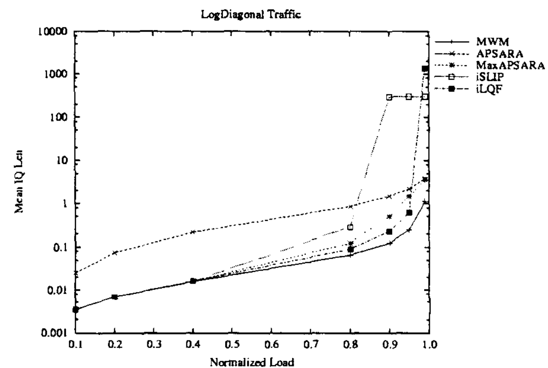

Fig. 4. Mean IQ length for logdiagonal traffic. Also in this case APSARA is the only algorithm able to approximate the MWM with bounded delays.

Table I reports the maximum achievable throughput for all the traffic considered.

C. Simulation results: Randomized variant

We next study the performance of the randomized variant of APSARA in order to understand the performance degradation due to the reduced number of neighbors ex-

Algorithm	Uniform	Logdiagonal	Diagonal
MWM	> 99%	> 99%	> 99%
APSARA	> 99%	> 99%	> 99%
iLQF	> 99%	≈ 97%	≈ 87%
iSLIP	> 99%	≈ 83%	≈ 82%

TABLE I

Maximum achievable throughput for different schedulers. APSARA is able to reach the same throughput as MWM under all traffic scenarios considered.

plored. We only consider the diagonal and logdiagonal loading (the randomized version performs well under uniform loading). We shall denote by APSARA-K the curves corresponding to the exploration of K neighbors. Thus, for $N = 32$, the basic version is denoted as APSARA-496 ($K = \binom{32}{2} = 496$). We consider two randomized versions: $K = N = 32$ and $K = \log_2 N = 5$, denoted by APSARA-32 and APSARA-5 respectively. Figure 5 and 6 show the mean IQ length for the diagonal and logdiagonal loading. As K decreases, of course the average delay increases; not by much under logdiagonal loading and by quite a bit under diagonal loading. Somewhat surprisingly, the throughput performance of APSARA with $K = \log N$ is quite good, upto 100%.

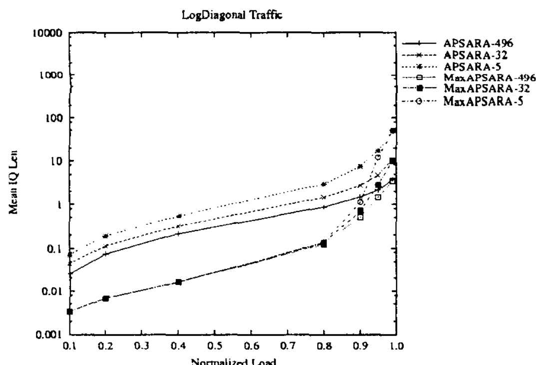

Fig. 5. Mean IQ length for logdiagonal traffic. Two randomized versions of APSARA are shown, with $|K| = N$ and $|K| = \log_2 N$, together with their maximal versions.

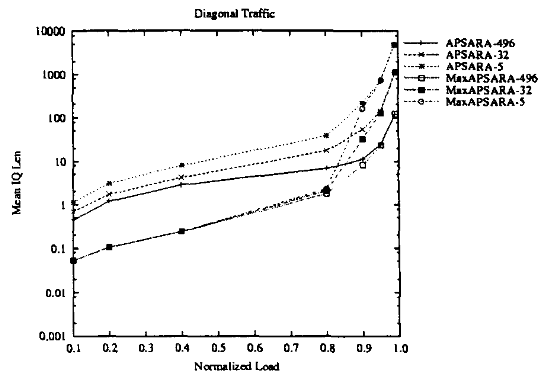

Fig. 6. Mean IQ length for diagonal traffic for the randomized versions of APSARA.

V. CONCLUSIONS

This paper proposes a switch scheduling algorithm called APSARA which exploits some specific features of the way packet switches operate to provide good performance while being simple to implement. APSARA achieves upto 100% throughput and simulations show that it provides very low average delays, comparable to that of the maximum weight matching. Since it uses hardware parallelism to search multiple candidate matchings, it only requires a single iteration. This makes it attractive for use in high-speed and/or large-sized switches. We have also explored a randomized version of the algorithm which searches a much smaller number of candidate matchings and have found the performance degradation to be quite small.

REFERENCES

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

[2] 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.

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

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

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

[6] 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.

[7] 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, 1997, pp. 20-28.

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