An Implementable Parallel Scheduler for Input-Queued Switches

THE APSARA ALGORITHM IS AN INPUT-QUEUED SWITCH SCHEDULER THAT USES LIMITED PARALLELISM TO FIND A MATCHING IN A SINGLE ITERATION, AS COMPARED TO THE $O(N^3)$ ITERATIONS OF THE MORE COMMON MAXIMUM-WEIGHT MATCHING ALGORITHM. APSARA ALSO ACHIEVES A THROUGHPUT OF UP TO 100 PERCENT AND HAS VERY GOOD DELAY PROPERTIES.

Paolo Giaccone
Politecnico di Torino

Devavrat Shah
Balaji Prabhakar
Stanford University

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

To perform well, however, an $N \times N$ IQ switch requires a good packet scheduling algorithm to determine which inputs to connect with which outputs in each time slot. The maximum-weight matching (MWM) algorithm finds, from among $N!$ possible matchings, the matching with the highest weight. 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 the oldest packet at input i for output j. The MWM algorithm is known to provide a 100 percent throughput[1-3] as long as no input or output is over subscribed. It achieves a low average delay by keeping queue lengths short. However, MWM is complex to implement: It needs $O(N^3)$ iterations in the worst case and does not lend itself to an easy pipelined implementation.

These implementation obstacles have motivated the proposal of several scheduling algorithms for high-speed switches, such as iterative SLIP (iSLIP),[4] iterative longest-queue first (iLQF),[5] reservation with preemption and acknowledgment (RPA),[6] and matrix unit cell scheduler (MUCS).[7] However, these algorithms perform poorly compared to MWM when the input traffic is nonuniform: They induce large delays, and their throughput can be less than 100 percent. Thus, although the other algorithms mentioned aim to solve implementation problems, their performance is poor.

This situation raises the question, is it possible for an algorithm to compete with MWM's performance 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:

- At most, packets arrive (or depart) at a rate of one per input (or output) port in a single given time slot. This means

0272-1732/02/$17.00 © 2002 IEEE

queue lengths, which MWM takes to be the weights, change very little during successive time slots. This situation suggests that a heavily weighted matching will continue to be heavily weighted for a few more time slots.

- Two randomly chosen matchings that differ by very few (for example, two) edges will quite likely be just as heavy. That is, given heavy matching M, there is quite likely matching M' that is a "neighbor" of M and 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.

When made precise, these observations yield the Apsara algorithm, which uses parallelism in hardware to search for a good matching in each time slot. More importantly, Apsara needs only one iteration per time slot, regardless of the switch's size.

Switch model

We consider an $N \times N$ input-queued switch, and partition the buffer at input i into N virtual output queues (VOQs). VOQ_{ij} stores packets at input i for output j. Following common practice, we assume fixed packet lengths. (Although Internet packet lengths vary, it is common for high-speed routers to fragment them into fixed-length cells before switching and to reassemble the cells into packets at the egress port.) We then can 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—that is, $\Sigma_j \lambda_{ij} \le 1$ for every i, and $\Sigma_i \lambda_{ij} \le 1$ for every j. In other words, this condition ensures that no input or output is oversubscribed.

We assume the switching fabric to be internally nonblocking (for example, we assume it to be 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 binary variables $x_{ij}(t)$, 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 output j even if $q_{ij}(t) = 0$. We can now model the crossbar constraint as follows:

$$x_{ij}(t) \in \{0, 1\};\ i, j = 1, \ldots, N$$

$$\sum_{j=1}^{N} x_{ij}(t) = 1,\ i = 1, \ldots, N$$

$$\sum_{i=1}^{N} x_{ij}(t) = 1,\ j = 1, \ldots, N$$

We can consider a feasible connection configuration as a matching in a bipartite graph. 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. For an $N \times N$ switch, the set of all possible matchings, denoted by S_N, has cardinality $N!$.

It is the job of the switch scheduling algorithm to determine, at each time t, the particular matching that it will use. 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 $(i + 1) \bmod N$, and so on. In this case, we can denote the corresponding matching matrices as $x_{ij}(t) = 1$ if $j = (i + t) \bmod N$ for every j, $1 \le j \le N$.

A scheduling algorithm of particular interest is MWM; we define it as follows. Denote the weight of matching $X(t) = [x_{ij}(t)]$ as $W(t) = \Sigma_{ij}\, q_{ij}(t)\, x_{ij}(t)$, taking the weight of the edge between input i and output j to be equal to queue length $q_{ij}(t)$.

MWM chooses, at each time t, the matching with the highest weight. More precisely, if $X^w(t)$ denotes the matching determined by MWM at time t, then

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

Researchers have shown that for all admissible independent and identically distributed Bernoulli input traffic patterns, MWM delivers up to 100 percent throughput.[1,2] Dai and

Prabhakar's work later relaxed the restriction of independent and identically distributed Bernoulli inputs.[3] Furthermore, extensive simulations show that MWM has low delays. However, MWM's main drawback is that it is difficult to implement in very high-speed and/or large switches. Our proposed algorithm alleviates these implementation problems.

Apsara

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.

- Queue lengths $q_{ij}(t)$ do not change much between iterations. Indeed, each $q_{ij}(t)$ can increase by one (at most) because of a possible arrival. They can also decrease by at most one because of a possible departure. This situation implies that a matching's weight changes by a bounded amount, making it likely that a heavy matching will tend to stay heavy over several time slots.
- We will call matchings X and Y that differ in very few edges neighbors, denoting by $N(X)$ the set of all neighbors of matching X. The observation we shall exploit is that if X is a heavy matching, it has a very good chance of having neighbor $X \in N(X)$ that is also heavy. If $N(X)$'s cardinality is small, we can conduct this process as a parallel search in hardware.

We will shortly more precisely define what we mean by a neighbor. For now, we note that the two features work in our favor in the following way: Given matching $X(t)$ at time t, we explore $N[X(t)]$ in parallel to determine if there is matching X' that is heavier than $X(t)$. If yes, we use the heaviest such matching at time $t + 1$. Otherwise, we continue to use matching $X(t)$ at time $t + 1$. The two observations we base this method on suggest that the weight of the matching at time $t + 1$ is likely to be quite good.

Given matching $X = [x_{ij}]$, let $\pi(i) = j$ if $x_{ij} = 1$. That is, matching X connects input i to output $\pi(i)$. This lets us shorten the matching's representation. For example, suppose $N = 3$; consider the matching

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

We can represent this matrix as vector $\{\pi(1), \pi(2), \pi(3)\} = \{2, 1, 3\}$.

Definition 1: Neighbor

We say that matching Y is a neighbor of matching X if and only if there are exactly two inputs—say i_1 and i_2—such that Y connects input i_1 to output $\pi(i_2)$ and also connects input i_2 to output $\pi(i_1)$. All other input-output pairs are the same under X and Y. X and Y differ in only two edges; the other $N - 2$ edges are the same for both. From a practical point of view, computation of a matching's neighbor is easily implemented in hardware.

Definition 2: Neighborhood set

We use $N(X)$ to denote the set of all neighbors of matching X. The cardinality of $N(X)$ is $N(N - 1)/2$.

As an example, consider a 3 × 3 switch. Matching X and its three neighbors X_1, X_2, and X_3 are

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

Hamiltonian walk on matchings

Before presenting the algorithm, we need to explain one last concept, that of a Hamiltonian walk on the set of all matchings. We introduce the walk and use it in describing the Apsara algorithm only because it lets us prove that Apsara achieves up to 100 percent throughput. In the section on simulations, we do not use this concept, and yet we shall find that Apsara achieves 100 percent 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 nodes, one after the other, exactly once in time $t = 1, \ldots, N!$. We extend $Z(t)$ for $t > N!$ by defining $Z(t) = Z[(t \bmod N!)+1]$. Nijenhuis and Wilf describe one simple algorithm for such a Hamiltonian walk.[8]

This algorithm produces $Z(t)$ such that, for

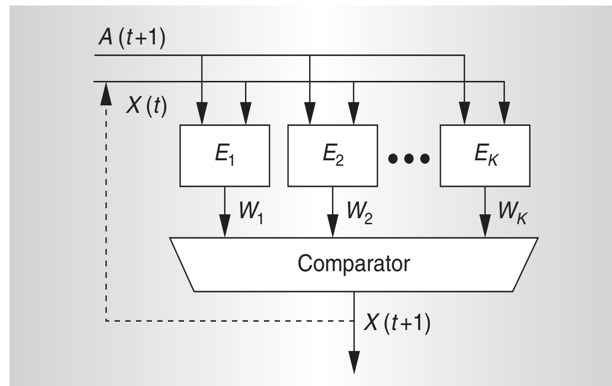

Figure 1. Schematic of Apsara implementation. Apsara uses old matching $X(t)$ and new arrivals $A(t+1)$ to compute the weights of neighbor matchings in parallel. E_i computes the weight of the ith neighbor of $x(t)$. This scheme determines new matching $X(t+1)$ as a result of weight comparison.

all t, $Z(t+1)$ is a neighbor of $Z(t)$. When we execute this algorithm for $N = 3$, it generates the following matchings: $Z(1) = (1, 2, 3)$, $Z(2) = (1, 3, 2)$, $Z(3) = (3, 1, 2)$, $Z(4) = (3, 2, 1)$, $Z(5) = (2, 3, 1)$, $Z(6) = (2, 1, 3)$, $Z(7) = Z(1)$, $Z(8) = Z(2)$, and so on.

Basic Apsara 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:

- Determine neighbors $N[X(t)]$ of $X(t)$ and matching $Z(t+1)$ that correspond to the Hamiltonian walk at time $t+1$.
- Let $S(t+1) = N[X(t)] \cup Z(t+1) \cup X(t)$. Compute the weight of every matching $Y \in S(t+1)$ as $W(Y) = \Sigma_{ij}\, y_{ij} q_{ij}\,(t+1)$.
- Determine the matching at time $t+1$ given by $X(t+1) = \arg\max_{U \in S(t+1)} \{W(U)\}$

This basic version of Apsara requires computing the weight of neighbor matchings. Each such computation is easy because neighbor Y differs from matching $X(t)$ in exactly two edges. However, computing the weights of all $N(N-1)/2$ 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: a few ports connected to very high-speed lines or several ports connected to lower-speed lines. So, if the goal is to build a high-aggregate-bandwidth switch with a few ports (say, 30 to 40), you require less than 800 modules for computing the weights of neighbor matchings. The big win in this case is in time (Apsara requires only one 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, you want to build a switch with say, 1,000 ports, you need up to 500,000 modules, which can be prohibitively expensive. We approach this issue from a different direction. Say that hardware space constraints allow the use of at most $K << N^2$ modules. How can we efficiently conduct the search procedure that Apsara requires?

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

Apsara randomized variant

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

- Pick K elements uniformly and at random from set $N[X(t)]$. Let $N_K[X(t)]$ denote the set of these elements. Note that it is not necessary to generate $N[X(t)]$. Determine matching $Z(t+1)$ that corresponds to the Hamiltonian walk at time $t+1$.
- Let $S_K(t+1) = N_K[X(t)] \cup Z(t+1) \cup X(t)$. For every matching $Y \in S_K(t+1)$, compute $W(Y) = \Sigma_{ij}\, y_{ij} q_{ij}\,(t+1)$.
- Then $X(t+1) = \arg\max_{U \in S_K(t+1)} \{W(U)\}$

We conclude our description of Apsara by mentioning one last point. Apsara generates all the matchings in the neighborhood set regardless of the current queue lengths. Apsara only

uses queue lengths to select the heaviest matching from the neighborhood set. For this reason, it is possible that the Apsara-determined matching could be heavy, but not of maximal size. That is, there exists an input, say i, which has packets for output j, but matching $X(t)$ connects input i to some other output j' and connects output j to some other input i'. Both $q_{ij'}(t)$ and $q_{i'j}(t)$ are also equal to 0. Thus, input i and output j will both idle unnecessarily.

If needed, it is easy to convert Apsara-determined matching $X(t)$ into a maximal matching. We call this maximal version Max-Apsara.

Apsara throughput theorem

We state the following theorem but do not provide a proof because of space limitations.

Theorem: Apsara is stable (it achieves up to 100 percent throughput) for any admissible independent and identically distributed Bernoulli packet arrival process.

Performance

We compare Apsara's performance with that of other algorithms: iSLIP and iLQF (both run with up to N iterations) and MWM. Recall that we do not use matching $Z(t)$ from the Hamiltonian walk. Although this version of Apsara should perform worse (because it has one less matching at its disposal), we shall show that even this version performs quite well, giving up to 100 percent throughput and acceptable delays.

Simulation settings

We first must define a particular switch, input traffic, and performance measures.

Switch. We use a switch with $N = 32$. Each VOQ can store up to 10,000 packets, and the switch drops excess packets.

Input traffic. We equally loaded all inputs on a normalized scale, and $\rho \in (0, 1)$ denotes the normalized load. The arrival process is an independent and identically distributed Bernoulli process. Let $|k| = (k \bmod N)$. We used the following load matrices to test Apsara's performance:

- *Uniform.* In a uniform matrix, $\lambda_{ij} = \rho/N$ $\forall\ i,j$. This is the most commonly used test traffic in the literature.
- *Diagonal.* A diagonal loading has $\lambda_{ii} = 2\rho/3$, $\lambda_{i|i+1|} = \rho/3\ \forall\ i$, and $\lambda_{ij} = 0$ for all other i and j. This loading is skewed in the sense that input i only has packet outputs i and $|i + 1|$. It is more difficult to schedule than uniform loading.
- *Log diagonal.* For a log-diagonal loading, $\lambda_{ij} = 2\lambda_{i|j+1|}$ and $\Sigma_i\ \lambda_{ij} = \rho$. For example, the distribution of 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, a specific algorithm's performance will become worse as the loading changes from uniform to log diagonal to diagonal.

Performance measures. We compared the performance of different algorithms by measuring the mean IQ lengths and computed delays directly using Little's formula.

The simulations ran until the estimate of the average delay reached the relative width of confidence interval equal to 1 percent with probability ≥ 0.95. We estimated confidence interval width by using the batch means approach, one standard technique to evaluate simulation results.

Simulation results for basic version

Figures 2 (next page) show the mean input queue lengths for uniform, diagonal, and log-diagonal loadings. Among all the algorithms considered, Apsara is the only nonmaximal 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 the other algorithms for low load. Max-Apsara can bridge the gap in the delays, and the effect of 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 can reach 100 percent throughput under all possible traffic loadings.

Table 1 (next page) summarizes the maximum achievable throughput for all the traffic considered.

Simulation results: Randomized variant

We next study the performance of Aspara's randomized variant to understand the per-

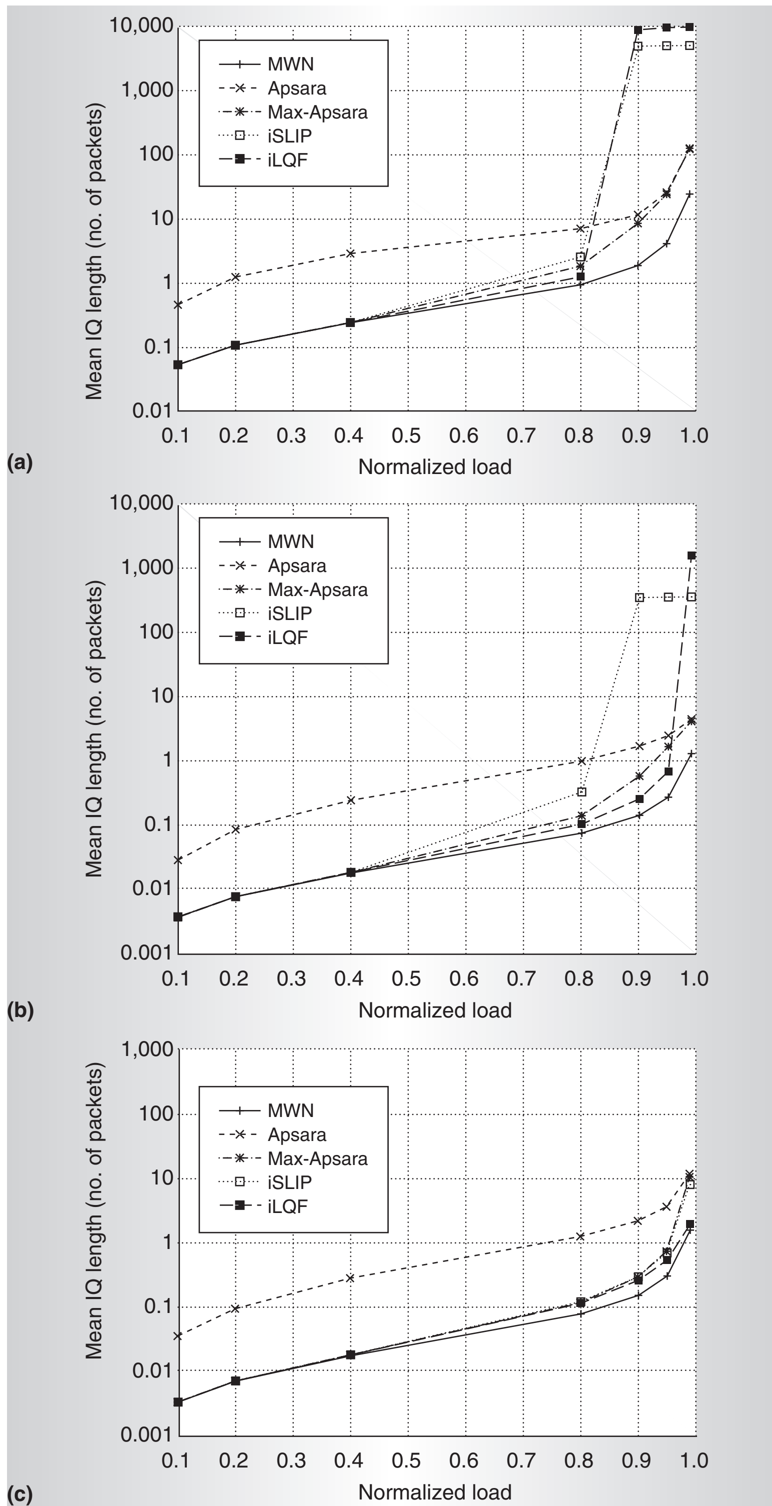

Figure 2. Mean IQ length for uniform traffic (a). Apsara is well behaved, but its queue occupancies are greater than those of the other algorithms for low loads, because it is not maximal . For diagonal traffic (b) and log-diagonal traffic (c), Apsara is the only algorithm able to approximate MWM with bounded delays.

Table 1. Maximum achievable throughput for different schedulers. Apsara can reach the same throughput as MWM under all traffic scenarios considered.

	Throughput for given loading (%)		
Algorithm	Uniform	Log diagonal	Diagonal
MWM	> 99	> 99	> 99
Apsara	> 99	> 99	> 99
iLQF	> 99	≈ 97	≈ 87
iSLIP	> 99	≈ 83	≈ 82

formance degradation because of the reduced number of neighbors explored. We only consider diagonal and log-diagonal loadings; the randomized version performs well under uniform loading. We use Apsara-K to denote the curves corresponding to the exploration of K neighbors. Thus, for $N = 32$, we denote the basic version Apsara-496 [$K = 32(32 - 1)/2 = 496$]. We also consider two randomized versions, $K = N = 32$ and $K = \log_2 N = 5$, denoting these versions Apsara-32 and Apsara-5. Figures 3 and 4 show the mean IQ length for log-diagonal and diagonal loadings. As K decreases, the average delay, of course, increases. The increase is slight under log-diagonal loading and quite a bit under diagonal loading. Somewhat surprisingly, the Aspara's throughput performance with $K = \log_2 N$ is quite good, up to 100 percent.

We can consider the proposed scheduling algorithm as a general framework to design practical, high-performance schedulers for input-queued switches. This framework is general enough to be used for designing hardware circuits for efficiently solving linear optimization problems approximately. MICRO

References

1. N. McKeown, V. Anantharan, and J. Walrand, "Achieving 100% Throughput in an Input-Queued Switch," *Proc. 15th Ann. Joint Conf. IEEE Computer and Comm. Societies* (Infocom 96), IEEE CS Press, Los Alamitos, Calif., 1996, pp. 296-302.
2. L. Tassiulas and A. Ephremides, "Stability Properties of Constrained Queueing Systems and Scheduling Policies for Maximum

Throughput in Multihop Radio Networks," *IEEE Trans. Automatic Control*, vol. 37, no. 12, Dec. 1992, pp. 1936-1948.
3. J. Dai and B. Prabhakar, "The Throughput of Data Switches with and without Speedup," *IEEE Infocom 2000*, IEEE Press, Piscataway, N.J., 2000, pp. 556-564.
4. N. McKeown, "iSLIP: A Scheduling Algorithm for Input-Queued Switches," *IEEE Trans. Networking*, vol. 7, no. 2, Apr. 1999, pp. 188-201.
5. N. McKeown, *Scheduling Algorithms for Input-Queued Cell Switches,* PhD thesis, CS/EE Dept., Univ. of California, Berkeley, 1995.
6. M. Ajmone Marsan et al., "RPA: A Flexible Scheduling Algorithm for Input Buffered Switches," *IEEE Trans. Communications*, vol. 47, no. 12, Dec. 1999, pp. 1921-1933.
7. H. Duan et al., "A High Performance OC12/OC48 Queue Design Prototype for Input Buffered ATM Switches," *INFOCOM 97: 16th Ann. Joint Conf. IEEE Computer and Comm. Societies* (Infocom 97), IEEE CS Press, Los Alamitos, Calif., 1997, pp. 20-28.
8. A. Nijenhuis and H. Wilf, *Combinatorial Algorithms: For Computers and Calculators,* 2nd ed., Academic Press, New York, 1978, p. 56.

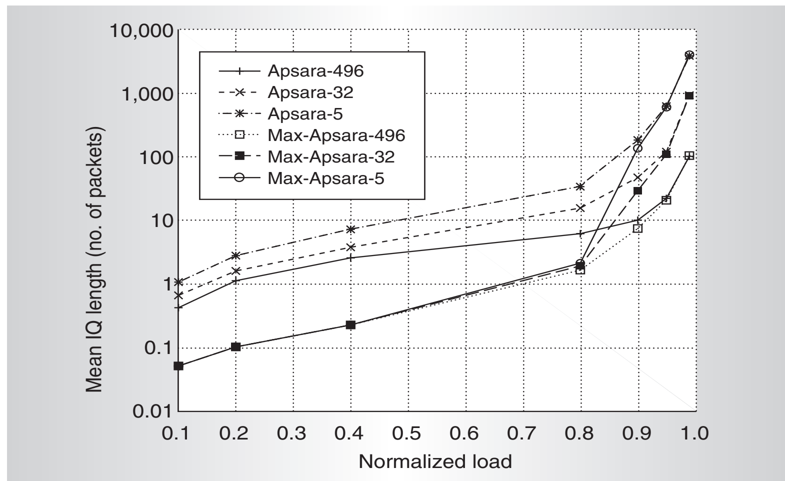

Figure 3. Mean IQ length for diagonal traffic for basic and randomized versions of Apsara. These randomized versions have $K = N$ and $|K|=\log_2 N$; the graphs also show results for their maximal versions.

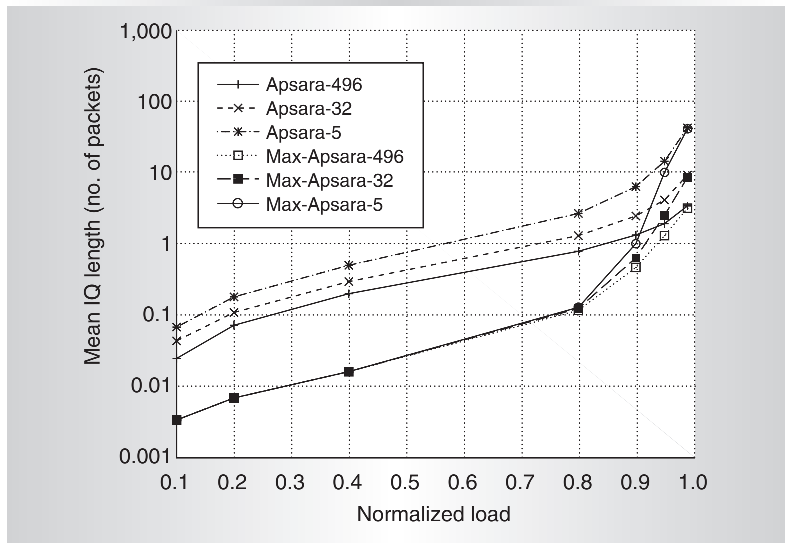

Figure 4. Mean IQ length for log-diagonal traffic for basic and randomized versions of Apsara.

Paolo Giaccone is a PhD student in the Electronics Department at Politecnico di Torino, Italy. His research interests include scheduling algorithms for input-queued switches. Giaccone has a DrIng in telecommunications engineering from Politecnico di Torino. He is a student member of the IEEE.

Devavrat Shah is a PhD candidate in the Department of Computer Science at Stanford University. His research interests include the design, analysis, and implementation of network algorithms. Shah has a bachelor's degree in computer science and engineering from the Indian Institute of Technology, Bombay.

Balaji Prabhakar is an assistant professor of electrical engineering and computer science and a Terman Fellow at Stanford University. He is also a fellow of the Alfred P. Sloan Foundation. His research interests include network algorithms (especially for switching, routing and quality of service), wireless networks, Web caching, network pricing, information theory, and stochastic network theory. Prabhakar has a PhD from the University of California at Los Angeles. He has received a Career award from the National Science Foundation and the Erlang Prize from the Informs Applied Probability Society.

Direct questions and comments about this article to Paolo Giaccone, Dipartimento di Elettronica, Politecnico di Torino, C.so Duca Abruzzi 24, 10129 Torino, Italy, giaccone@polito.it.