[24] McKeown, N; “Scheduling Algorithms for Input-Queued Cell Switches,” PhD Thesis, University of California at Berkeley, 1995.

[25] McKeown, N; Anantharam, V; and Walrand, J; “Achieving 100% Throughput in an Input-Queued Switch,” *Proceedings of IEEE Infocom ‘96*, San Francisco, March 1996.

[26] McKeown, N.; Izzard, M.; Mekkittikul, A.; Ellersick, W.; and Horowitz, M.; “The Tiny Tera: A Packet Switch Core” *Hot Interconnects V, Stanford University*, August 1996.

[27] McKeown, N.; Prabhakar, B., and Zhu, M.; “Matching Output Queueing with Combined Input and Output Queueing,” Proceedings of the Allerton Conference, Allerton, September 1997.

[28] Mekkittikul, A.; McKeown, N.; “A Fair Algorithm For Achieving 100% Throughput in an Input-Queued Switch,” *Proceedings of IC3N ‘96*, Maryland, October 1996.

[29] Newman, P.; Lyon, T.; and Minshall, G.; “Flow Labelled IP: A Connectionless Approach to ATM,” *Proceedings of IEEE INFOCOM ‘96*, San Francisco, March 1996.

[30] Obara, H.; “Optimum architecture for input queueing ATM switches,” *IEE Electronics Letters*, pp.555-557, 28th March 1991.

[31] Partridge, C., et al. “A fifty gigabit per second IP router,” To appear in *IEEE/ACM Transactions on Networking*.

[32] Parulkar, G; Schmidt, D.; Turner, J.; “AITPM: A Strategy for Integrating IP with ATM,” *Proceedings of IEEE INFOCOM ‘96*, San Francisco, March 1996.

[33] Tamir, Y.; and Chi, H-C.;“Symmetric Crossbar Arbiters for VLSI Communication Switches,” *IEEE Trans on Parallel and Dist. Systems*, Vol. 4, No.1, pp.13-27, 1993.

[34] Tamir, Y.; Frazier, G.; “High performance multi-queue buffers for VLSI communication switches,” *Proc. of 15th Ann. Symp. on Comp. Arch.*, pp.343-354, June 1988.

[35] Tarjan, R.E.; “Data structures and network algorithms,” *Society for Industrial and Applied Mathematics*, Pennsylvania, Nov 1983.

[36] Troudet, T.P.; Walters, S.M.; “Hopfield neural network architecture for crossbar switch control,” *IEEE Trans. Circuits and Systems*, Vol.CAS-38, pp.42-57, Jan.1991.

[37] Chang, C-Y.; Paulraj, A.J.; Kailath, T.; “A Broadband Packet Switch Architecture with Input and Output Queueing,” *Proc. Globecom ‘94*, pp.448-452.

[38] Iliadis, I.; Denzel, W.E.; “Performance of packet switches with input and output queueing,” *Proceedings of ICC `90*, Atlanta, GA, pp.747-53, April 1990.

[39] Gupta, A.L.; Georganas, N.D.; “Analysis of a packet switch with input and output buffers and speed constraints,” *Proceedings of Infocom ‘91*, Bal Harbour, FL, pp.694-700, April 1991.

[40] Y. Oie; M. Murata, K. Kubota, and H. Miyahara, \E ect of speedup in nonblocking packet switch,” *Proceedings of ICC `89*, Boston, MA, pp. 410-14, June 1989.

[41] Chen, J. S-C.; Stern, T.E.; “Throughput analysis, optimal buffer allocation, and traffic imbalance study of a generic nonblocking packet switch,” *IEEE Journal of Selected Areas in Communications*, Vol. 9, No. 3, pp. 439-49, April 1991.

11 Acknowledgements

Along the way, the development of *i*SLIP was helped by discussions with Tom Anderson, Richard Edell, Jean Walrand and Pravin Varaiya, all at the University of California at Berkeley. The gate counts shown in Section 9 were obtained by Pankaj Gupta at Stanford University.

12 References

[1] Ali, M.; Nguyen, H.; "A neural network implementation of an input access scheme in a high-speed packet switch," *Proc. of GLOBECOM 1989*, pp.1192-1196. 1989.

[2] Anderson, T.; Owicki, S.; Saxe, J.; and Thacker, C.; "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems.* pp. 319-352. November 1993.

[3] Anick, D.; Mitra, D.; Sondhi, M.M.; "Stochastic theory of a data-handling system with multiple sources," *Bell System Technical Journal*, Vol.61, pp.1871-1894, 1982.

[4] Ascend Communications GRF IP SwitchTechincal Product Description, http://www.ascend.com/230.html.

[5] Brown, T.X; Liu, K.H.; "Neural network design of a Banyan network controller," *IEEE J. Selected Areas Communications*, Vol.8, pp.1289-1298, Oct. 1990.

[6] Cisco Systems GSR 12000 Technical Product Description, http://www.cisco.com/warp/public/733/12000/index.shtml.

[7] Rekhter, Y.; Davie, B.; Katz, D.; Rosen, E.; Swallow, G.; "Cisco Systems' Tag Switching Architecture Overview", Internet RFC 2105, October 1997. http://info.internet.isi.edu/in-notes/rfc/files/rfc2105.txt.

[8] Chen, M.; Georganas, N.D.; "A fast algorithm for multi-channel/port traffic scheduling" *Proc. IEEE Supercom/ICC '94*, pp.96-100.

[9] Chiussi, F.M. ; Tobagi, F.A.; "Implementation of a Three-Stage Banyan-Based Architecture with Input and Output Buffers for Large Fast Packet Switches," *Stanford CSL Techical Report CSL-93-577*, June 1993.

[10] Cruz, R.; "A calculus for network delay, part I: network elements in isolation," *IEEE Trans. Information Theory*, Vol. 37, No.1, pp.114-121, 1991.

[11] Heffes, H.; Lucantoni, D. M.; "A Markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance," *IEEE J. Selected Areas in Communications*, vol.4, pp.856-868. 1988.

[12] Hopcroft, J.E.; Karp, R.M.; "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *Society for Industrial and Applied Mathematics Journal of Computation*, Vol. 2, pp.225-231. 1973.

[13] Huang, A.; Knauer, S.; "Starlite: A wideband digital switch," *Proc. GLOBECOM '84* (1984), pp.121-125.

[14] Hui, J.; Arthurs, E.; "A broadband packet switch for integrated transport," *IEEE J. Selected Areas Communications, vol.5*, no. 8, pp. 1264-1273. Oct 1987.

[15] Jain, R.; Routhier, S.A.; "Packet Trains: measurements and a new model for computer network traffic," *IEEE J. Selected Area Communications*, Vol.4, pp.986-995, 1986.

[16] Karol, M.; Hluchyj, M.; and Morgan, S.; "Input versus output queueing on a space division switch," *IEEE Trans. Communications*, vol. 35, no.12, pp.1347-1356, 1988.

[17] Karol, M.; Hluchyj, M.; "Queueing in high-performance packet-switching," *IEEE J. Selected Area Communications*, Vol.6, pp.1587-1597, Dec. 1988.

[18] Karol, M.; Eng, K.; Obara, H.; "Improving the performance of input-queued ATM packet switches," *INFOCOM '92*, pp.110-115. March 1992.

[19] LaMaire, R.O.; Serpanos, D.N.; "Two-dimensional round-robin schedulers for packet switches with multiple input queues," *IEEE/ACM Transactions on Networking*, vol.2, (no.5), pp.471-82. October 1993.

[20] Low, S.; Varaiya, P.; "Burstiness bounds for some burst reducing servers," *Proc. INFOCOM '93*, pp.2-9, March 1993.

[21] Kesidis, G.; Walrand, J.; Chang, C.-S.; "Effective bandwidths for multiclass Markov fluids and other ATM sources." *IEEE/ACM Transactions on Networking*, vol.1, (no.4):424-8, August 1993.

[22] Leland, W.E.; Willinger, W.; Taqqu, M.; Wilson, D.; "On the self-similar nature of Ethernet traffic", *Proc. of Sigcomm*, San Francisco, pp.183-193. Sept 1993.

[23] Lund, C.; Phillips, S.; Reingold, N.; "Fair prioritized scheduling in an input-buffered switch," *Proceedings of the IFIP-IEEE Conf. on Broadband Communications '96*, Montreal, pp. 358-69, April 1996.

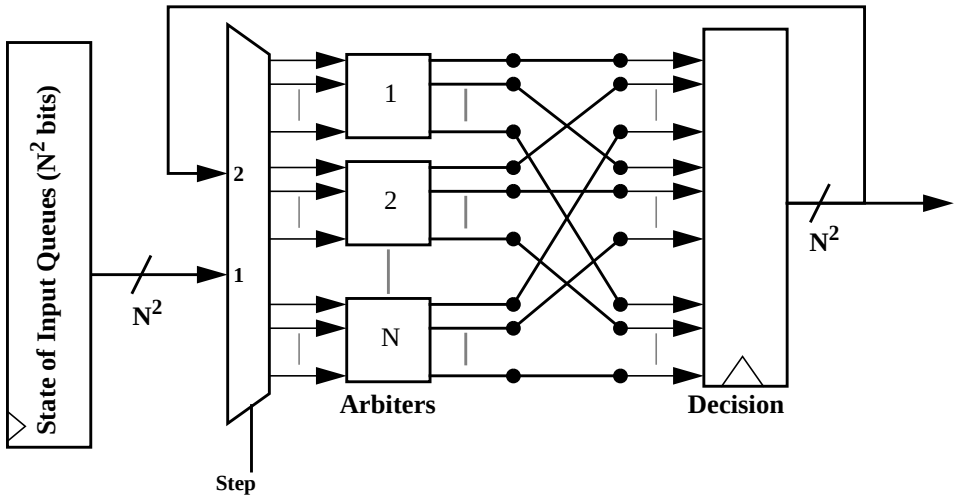


FIGURE 22 Interconnection of N arbiters to implement *i*SLIP for an $N \times N$ switch. Each arbiter is used for both input and output arbitration. In this case, each arbiter contains *two* registers to hold pointers g_i and a_i.

10 Conclusion

The Internet requires fast switches and routers to handle the increasing congestion. One emerging strategy to achieve this is to merge the strengths of ATM and IP; building IP routers around high speed cell switches. Current cell switches can employ shared output queueing due to relatively low bandwidths. Unfortunately, the growth in demand for bandwidth far exceeds the growth in memory bandwidth, making it inevitable that switches will maintain queues at their inputs. We believe that these switches will use virtual output queueing, and hence will need fast, simple, fair and efficient scheduling algorithms to arbitrate access to the switching fabric.

To this end, we have introduced the *i*SLIP algorithm; an iterative algorithm that achieves high throughput, yet is simple to implement in hardware and operate at high speed. By using round-robin arbitration, *i*SLIP provides fair access to output lines and prevents starvation of input queues. By careful control of the round-robin pointers, the algorithm can achieve 100% throughput for uniform traffic. When the traffic is non-uniform, the algorithm quickly adapts to an efficient round-robin policy among the busy queues. The simplicity of the algorithm allows the arbiter for a 32-port switch to be placed on single chip, and to make close to 100million arbitration decisions per second.

We are preparing a second paper that focusses on the implementation of the algorithm, but it suffices here to make the following observations. First, the area required to implement the scheduler is dominated by the 2*N* pro-

Switch Size (N)	Number of inverter equivalents per arbiter	Total number of inverter equivalents for *N* arbiters
4	274	2,194
8	384	6,148
16	642	20,560
32	1,210	77,440
64	2,420	154,848
128	4,591	587,648

Table 1 Number of inverter equivalents required to implement 1 and *N* arbiters for a prioritized-*i*SLIP scheduler, with four levels of priority.

grammable priority encoders. The number of inverter equivalents required to implement the programmable priority encoders for prioritized-*i*SLIP is shown in Table 1.[1] The number of gates for a 32-port scheduler is less than 100,000 making it readily implement in current CMOS technologies, and the total number of gates grows approximately with N^2. We have observed in two implementations that the regular structure of the design makes routing relatively straightforward. Finally, we have observed that the complexity of the implementation is (almost) independent of the number of iterations. When multiple iterations are used, the number of arbiters remain unchanged. The control overhead necessary to implement multiple iterations is very small.

In some implementations, it may be desirable to reduce the number of arbiters, sharing them among both the grant and accept steps of the algorithm. Such an implementation requiring only *N* arbiters[2] is shown in Figure 22. When the results from the grant arbiter have settled, they are registered and fed back to the input for the second step. Obviously each arbiter must maintain a separate register for the g_i and a_i pointers, selecting the correct pointer for each step.

1.These values were obtained from a VHDL design that was synthesized using the Synopsis design tools, and compiled for the Texas Instruments TSC5000 0.25µm CMOS ASIC process. The values for regular *i*SLIP will be smaller.

2.A slight performance penalty is introduced by registering the output of the grant step and feeding back the result as the input to the accept step. This is likely to be small in practice.

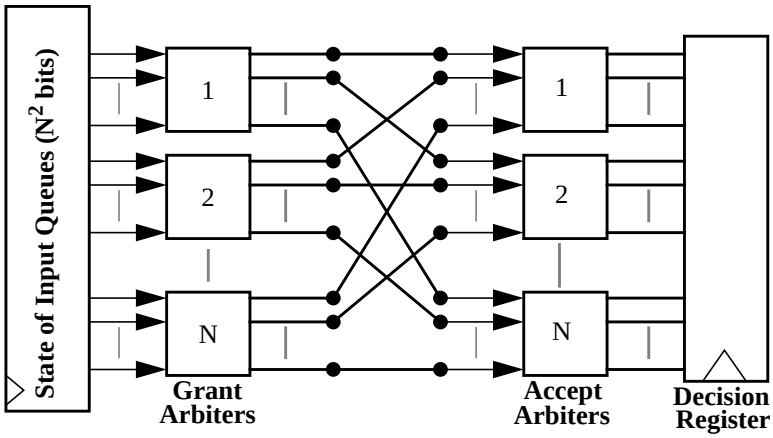


FIGURE 21 Interconnection of 2N arbiters to implement *i*SLIP for an $N \times N$ switch.

Figure 21 shows how 2*N* arbiters (*N* at each input and *N* at each output) and an N^2-bit memory are interconnected to construct an *i*SLIP scheduler for an $N \times N$ switch. The state memory records whether an input queue is empty or non-empty. From this memory, an N^2-bit wide vector presents *N* bits to each of *N grant* arbiters, representing **Step 1**—*Request*. The grant arbiters select a single input among the contending requests, thus implementing **Step 2**—*Grant*. The grant decision from each grant arbiter is then passed to the *N accept* arbiters, where each arbiter selects at most one output on behalf of an input, implementing **Step 3**—*Accept*. The final decision is then saved in a decision register and the values of the g_i and a_i pointers are updated. The decision register is used to notify each input which cell to transmit and to configure the crossbar switch.

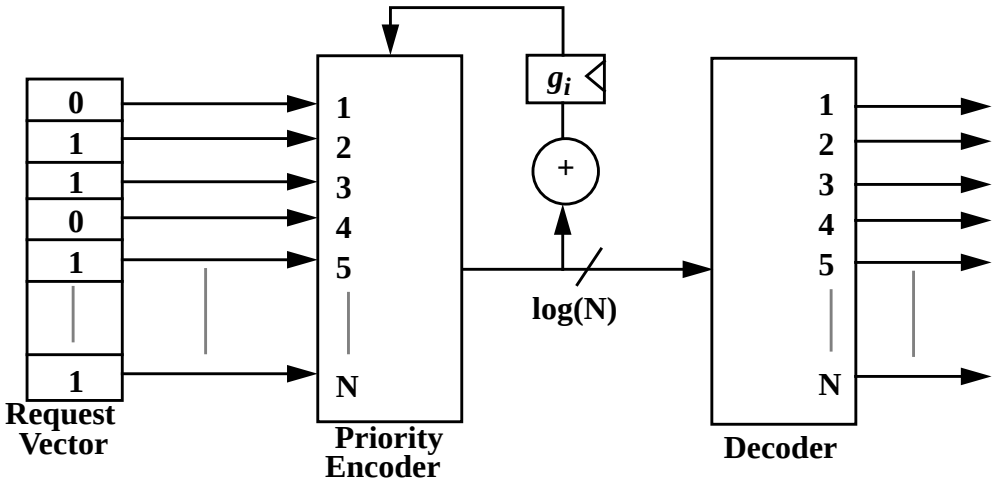


FIGURE 20 Round-robin *grant* arbiter for *i*SLIP algorithm. The priority encoder has a programmed highest-priority, g_i. The *accept* arbiter at the input is identical.

As illustrated in Figure 20, each *i*SLIP arbiter consists of a priority encoder with a programmable highest priority, a register to hold the highest priority value, and an incrementer to move the pointer after it has been updated. The decoder indicates to the next bank of arbiters which request was granted.

8.2 Threshold *i*SLIP

Scheduling algorithms that find a maximum *weight* match outperform those that find a maximum *sized* match. In particular, if the weight of the edge between input *i* and output *j* is the occupancy $L_{i,j}(t)$ of input queue *Q(i,j)* then we will conjecture that the algorithm can achieve 100% throughput for all i.i.d. Bernoulli arrival patterns. But maximum weight matches are significantly harder to calculate than maximum sized matches [35] and to be practical, must be implemented using an upper limit on the number of bits used to represent the occupancy of the input queue.

In the Threshold *i*SLIP algorithm we make a compromise between the maximum sized match and the maximum weight match by quantizing the queue occupancy according to a set of threshold levels. The threshold level is then used to determine the priority level in the Priority *i*SLIP algorithm. Each input queue maintains an ordered set of threshold levels $\mathbf{T} = \{t_1, t_2, \ldots, t_T\}$, where $t_1 < t_2 < \ldots < t_T$. If $t_a \le Q(i,j) < t_{a+1}$ then the input makes a request of level $l_i = a$.

8.3 Weighted *i*SLIP

In some applications, the strict priority scheme of Prioritized *i*SLIP may be undesirable, leading to starvation of low-priority traffic. The Weighted *i*SLIP algorithm can be used to divide the throughput to an output non-uniformly among competing inputs. The bandwidth from input *i* to output *j* is now a ratio $f_{ij} = \frac{n_{ij}}{d_{ij}}$ subject to the admissibility constraints $\sum_i f_{ij} < 1, \sum_j f_{ij} < 1$.

In the basic *i*SLIP algorithm each arbiter maintains an ordered circular list, $\boldsymbol{S} = \{1, \ldots, N\}$. In the Weighted *i*SLIP algorithm the list is expanded at output *j* to be the ordered circular list $\boldsymbol{S}_j = \{1, \ldots, W_j\}$ where $W_j = \text{LowestCommonMultiple}(d_{ij})$ and input *i* appears $\frac{n_{ij}}{d_{ij}} \times W_j$ times in $\boldsymbol{S}_j$.

9 Implementing *i*SLIP

An important objective is to design a scheduler that is simple to implement. To conclude our description of *i*SLIP, we consider the complexity of implementing *i*SLIP in hardware. We base our discussion on single-chip versions of *i*SLIP that have been implemented for 16-port [6] and 32-port [26] systems.

8 Variations on *i*SLIP

8.1 Prioritized *i*SLIP

Many applications use multiple classes of traffic with different priority levels. The basic *i*SLIP algorithm can be extended to include requests at multiple priority levels with only a small performance and complexity penalty. We call this the Prioritized *i*SLIP algorithm.

In Prioritized *i*SLIP each input now maintains a separate FIFO *for each priority level* and for each output. This means that for an $N \times N$ switch with *P* priority levels, each input maintains $P \times N$ FIFOs. We shall label the queue between input *i* and output *j* at priority level *l*, $Q_l(i, j)$ where $1 \le i, j \le N$, $1 \le l \le P$. As before, only one cell can arrive in a cell time, so this does not require a processing speedup by the input.

The Prioritized *i*SLIP algorithm gives *strict* priority to the highest priority request in each cell time. This means that $Q_l(i, j)$ will only be served if all queues $Q_m(i, j)$, $l < m \le P$ are empty.

The *i*SLIP algorithm is modified as follows:

Step 1. *Request*. Input *i* selects the highest priority non-empty queue for output *j*. The input sends the priority level l_{ij} of this queue to the output *j*.

Step 2. *Grant*. If output *j* receives any requests, it determines the highest level request. i.e. it finds $L(j) = \max_i(l_{ij})$. The output then chooses one input among only those inputs that have requested at level $L(j)$. The output arbiter maintains a separate pointer, g_{jl} for each priority level. When choosing among inputs at level *L(j)*, the arbiter uses the pointer $g_{jL(j)}$ and chooses using the same round-robin scheme as before. The output notifies each input whether or not its request was granted. The pointer $g_{jL(j)}$ is incremented (modulo *N*) to one location beyond the granted input if and only if input *i* accepts output *j* in step 3 of the first iteration.

Step 3. *Accept*. If input *i* receives any grants, it determines the highest level grant. i.e. it finds $L'(i) = \max_j(l_{ij})$. The input then chooses one output among only those that have requested at level $l_{ij} = L'(i)$. The input arbiter maintains a separate pointer, a_{il} for each priority level. When choosing among outputs at level $L'(i)$, the arbiter uses the pointer $a_{iL'(i)}$ and chooses using the same round-robin scheme as before. The input notifies each output whether or not its grant was accepted. The pointer $a_{iL'(i)}$ is incremented (modulo *N*) to one location beyond the accepted output.

Implementation of the Prioritized *i*SLIP algorithm is more complex than the basic *i*SLIP algorithm, but can still be fabricated from the same number of arbiters.

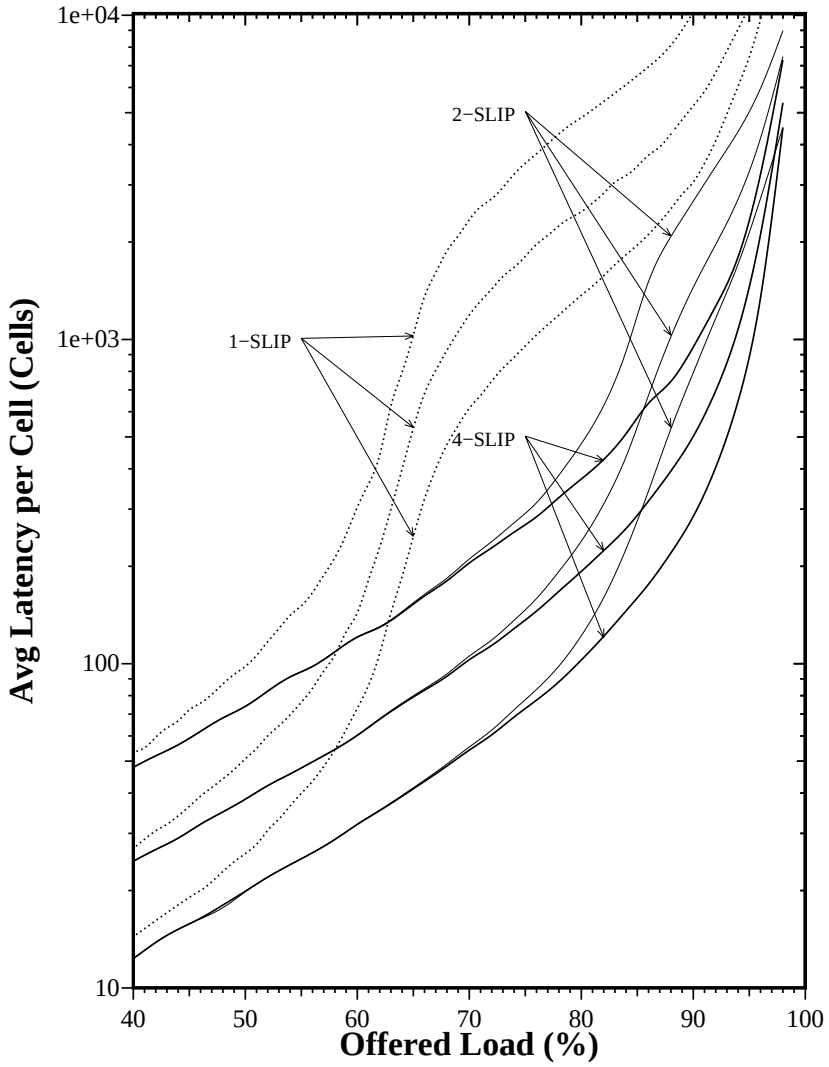


FIGURE 19 Performance of *i*SLIP for 1, 2 and 4 iterations under bursty arrivals. Arrival process is a 2-state Markov-modulated on-off process. Average burst lengths are 16, 32 and 64 cells.

latency is *proportional* to the expected burst length. The performance for bursty traffic is not heavily influenced by the queueing policy.

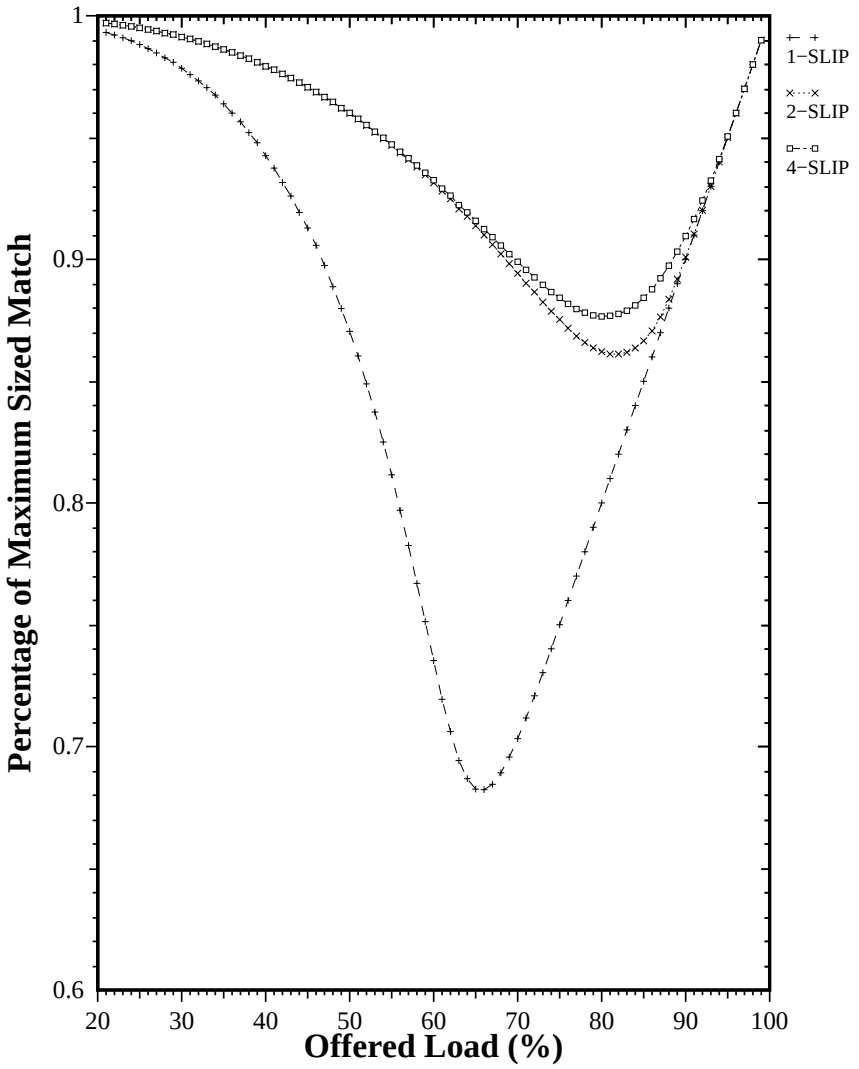


FIGURE 18 Comparison of the match size for *i*SLIP with the size of a maximum sized match for the same set of requests. Results are for a 16×16 switch and uniform i.i.d. Bernoulli arrivals.

mum sized match. But only up to a point: for an 16×16 switch under this traffic load, increasing the number of iterations beyond four does not measurably increase the average match size.

7.3 With Bursty Arrivals

We illustrate the effect of burstiness on *i*SLIP using an on-off arrival process modulated by a 2-state Markov-chain. Figure 19 shows the performance of *i*SLIP under this arrival process for a 16×16 switch, comparing the performance for 1, 2 and 4 iterations. As we would expect, the increased burst size leads to a higher queueing delay whereas an increased number of iterations leads to a lower queueing delay. In all three cases, the average

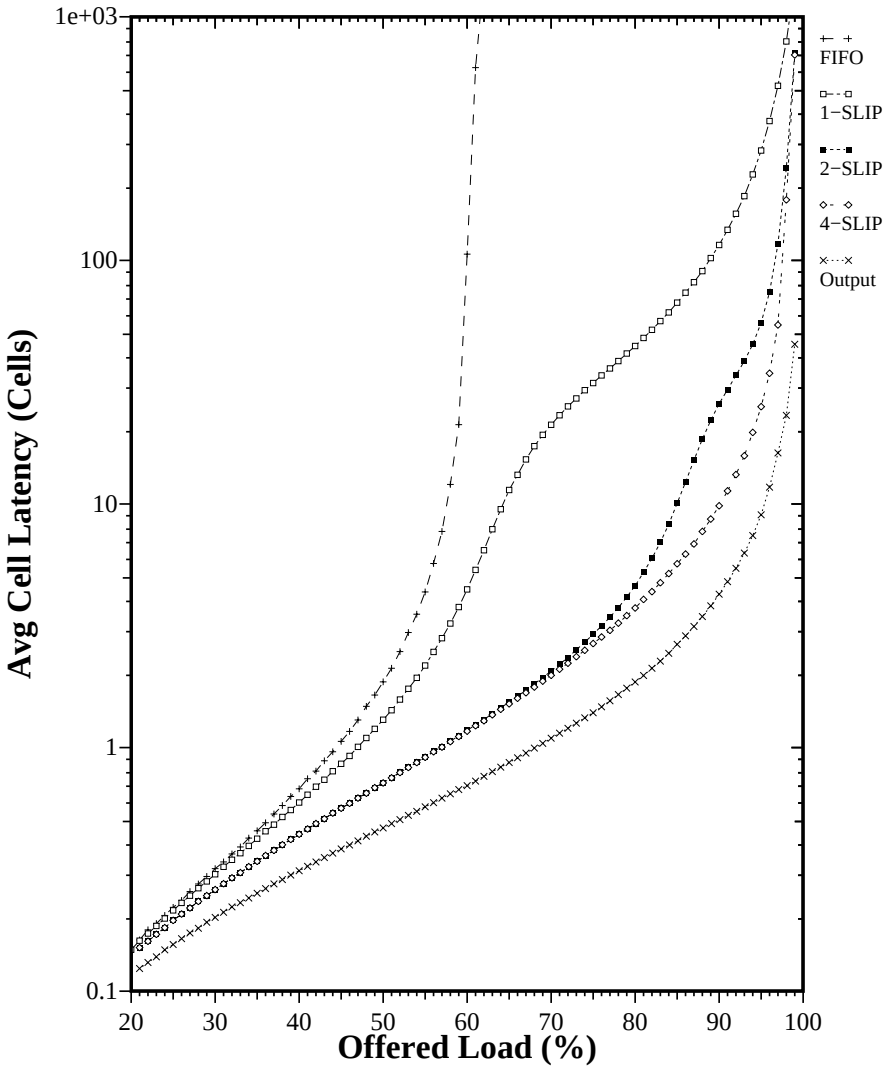


FIGURE 17 Performance of *i*SLIP for 1,2 and 4 iterations compared with FIFO and output queueing for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16×16 switch. The graph shows the average delay per cell, measured in cell times, between arriving at the input buffers and departing from the switch.

multiple iterations of *i*SLIP significantly increase the size of the match and therefore reduces the queueing delay. In fact, *i*SLIP can achieve 100% throughput for one or more iteration with uniform i.i.d. Bernoulli arrivals. Intuitively, the size of the match increases with the number of iterations: each new iteration potentially adds connections not made by earlier iterations. This is illustrated in Figure 18 which compares the size of *i*SLIP matching with the size of the maximum matching for the same instantaneous queue occupancies. Under low offered load, the *i*SLIP arbiters move randomly and the ratio of the match size to the maximum match size decreases with increased offered load. But when the load exceeds approximately 65%, the ratio begins to increase linearly. As expected, the ratio increases with the number of iterations indicating that the matching gets closer to the maxi-

Cell 1, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \ i_1 A\, j_1$$

Iteration N:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix}$$

Cell 2, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \end{bmatrix}$$

Iteration N-1:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \\ j_3 \\ : \\ j_N \end{bmatrix}$$

:

:

Cell N, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_N \\ i_{N-1} \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_N \\ j_{N-1} \\ : \\ j_1 \end{bmatrix}$$

FIGURE 16 Example of the number of iterations required to converge for a heavily loaded $N \times N$ switch. All input queues remain non-empty for the duration of the example. In the first cell time, the arbiters are all synchronized. During each cell time, one more arbiter is desynchronized from the others. After N cell times, all arbiters are desynchronized and a maximum sized match is found in a single iteration.

Property 4. The algorithm will converge in at most N iterations. Each iteration will schedule zero, one or more connections. If zero connections are scheduled in an iteration then the algorithm has converged: no more connections can be added with more iterations. Therefore, the slowest convergence will occur if exactly one connection is scheduled in each iteration. At most N connections can be scheduled (one to every input and one to every output) which means the algorithm will converge in at most N iterations.

Property 5. The algorithm will not necessarily converge to a maximum sized match. At best, it will find a *maximal* match: the largest size match without removing connections made in earlier iterations.

7 Simulated Performance of Iterative *i*SLIP

7.1 How Many Iterations?

When implementing *i*SLIP with multiple iterations, we need to decide how many iterations to perform during each cell time. Ideally, from Property 4 above we would like to perform N iterations. However, in practice there may be insufficient time for N iterations, and so we need to consider the penalty of performing only i iterations, where $i < N$. In fact, because of the desynchronization of the arbiters, *i*SLIP will usually converge in fewer than N iterations. An interesting example of this is shown in Figure 16. In the first cell time, the algorithm takes N iterations to converge, but thereafter converges in one less iteration each cell time. After N cell times, the arbiters have become totally desynchronized and the algorithm will converge in a single iteration.

How many iterations should we use? it clearly doesn't always take N. One option is to always run the algorithm to completion, resulting in a scheduling time that varies from cell to cell. In some applications this may be acceptable. In others, such as in an ATM switch, it is desirable to maintain a fixed scheduling time and to try and fit as many iterations into that time as possible.

Under simulation, we have found that for an $N \times N$ switch it takes *about* $\log_2 N$ iterations for *i*SLIP to converge. This is similar to the results obtained for PIM in [2], in which the authors prove that

$$E[I] \leq \log_2 N + \frac{4}{3}, \tag{4}$$

where I is the number of iterations that PIM takes to converge. For all the stationary arrival processes we have tried $E[I] < \log_2 N$ for *i*SLIP. However, we have not been able to prove that this relation holds in general.

7.2 With Benign Bernoulli Arrivals

To illustrate the improvement in performance of *i*SLIP when the number of iterations is increased, Figure 17 shows the average queueing delay for 1, 2 and 4 iterations under uniform i.i.d. Bernoulli arrivals. We find that

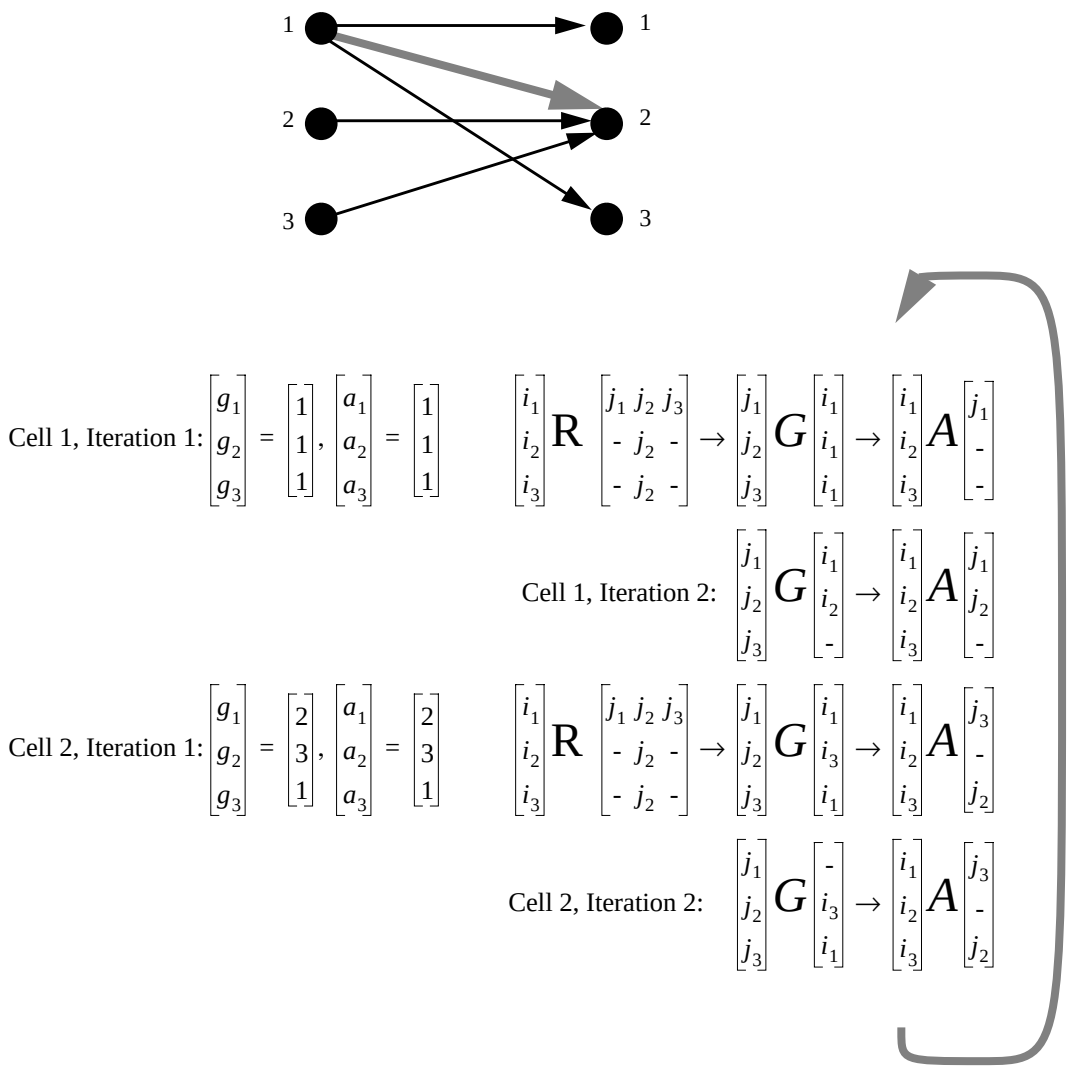


FIGURE 15 Example of starvation, if pointers are updated after every iteration. The 3×3 switch is heavily loaded, i.e. all active connections have an offered load of 1 cell per cell time. The sequence of grants and accepts repeats after 2 cell times, even though the (highlighted) connection from input 1 to output 2 has not been made. Hence, this connection will be starved indefinitely.

6.2 Properties

With multiple iterations, the *i*SLIP algorithm has the following properties:

Property 1. Connections matched in the first iteration become the lowest priority in the next cell time.

Property 2. No connection is starved. Because pointers are not updated after the first iteration, an output will continue to grant to the highest priority requesting input until it is successful.

Property 3. For *i*SLIP with more than one iteration, and under heavy load, queues with a common output may each have a different throughput. repeats every three cell times.

Step 2. *Grant*. If an unmatched output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo *N*) to one location beyond the granted input if and only if the grant is accepted in Step 3 *of the first iteration.*

Step 3. *Accept*. If an unmatched input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo *N*) to one location beyond the accepted output.

6.1 Updating Pointers

Note that pointers g_i and a_i are only updated for matches found in the first iteration. Connections made in subsequent iterations do not cause the pointers to be updated. This is to avoid starvation. To understand how starvation can occur, we refer to the example of a 3×3 switch with 5 active and heavily loaded connections, shown in Figure 15. The switch is scheduled using two iterations of the *i*SLIP algorithm, except in this case the pointers are updated after *both* iterations. The figure shows the sequence of decisions by the grant and accept arbiters; for this traffic pattern, they form a repetitive cycle in which *the highlighted connection from input 1 to output 2 is never served*. Each time the round-robin arbiter at output 2 grants to input 1, input 1 chooses to accept output 1 instead.

Starvation is eliminated if the pointers are not updated after the first iteration. In the example, output 2 would continue to grant to input 1 with highest priority until it is successful.

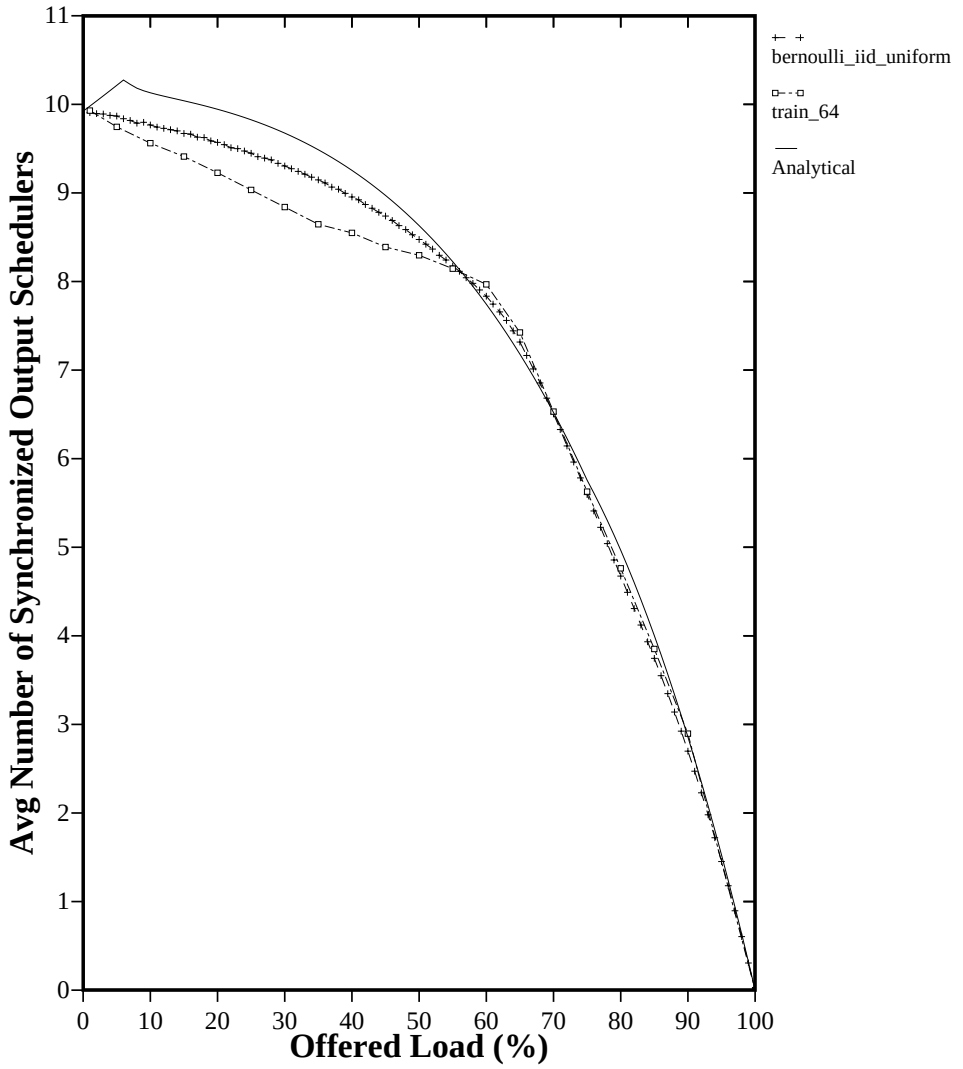


FIGURE 14 Comparison of analytical approximation and simulation results for the average number of synchronized output schedulers. Simulation results are for a 16×16 switch with i.i.d Bernoulli arrivals and an on-off process modulated by a 2-state Markov chain with an average burst length of 64 cells. The analytical approximation is shown in Equation 3.

improves as we increase the number of iterations (up to about log_2N, for an $N \times N$ switch). Once again, we shall see that *desynchronization* of the output arbiters plays an important rôle in achieving low latency.

When multiple iterations are used, it is necessary to modify the *i*SLIP algorithm. The three steps of each iteration operate in parallel on each output and input and are as follows:

Step 1. *Request.* Each unmatched input sends a request to every output for which it has a queued cell.

not constant: when a queue changes between empty and non-empty, the scheduler must adapt to the new set of queues that require service. This adaptation takes place over many cell times while the arbiters desynchronize again. During this time, the throughput will be worse than for the M/D/1 queue and the queue length will increase. This in turn will lead to an increased latency.

5.2 Desynchronization of Arbiters

We have argued that the performance of *i*SLIP is dictated by the degree of synchronization of the output schedulers. In this section we present a simple model of synchronization for a stationary and sustainable uniform arrival process.

In [24, see Appendix 1] we find an approximation for $\mathrm{E}[S(t)]$, the expected number of synchronized output schedulers at time *t*. The approximation is based on two assumptions:

1. Inputs that are unmatched at time *t* are uniformly distributed over all inputs.
2. The number of unmatched inputs at time *t* has zero variance.

This leads to the approximation

$$\mathrm{E}[S(t)] \approx N - \lambda N\left(\frac{\lambda N - 1}{\lambda N}\right)^{\lambda\bar{\lambda}N} - \bar{\lambda}^2 N\left(\frac{\bar{\lambda}N - 1}{\bar{\lambda}N}\right)^{\bar{\lambda}^2 N - 1} \tag{3}$$

where,

N = number of ports,
λ = arrival rate averaged over all inputs,
$\bar{\lambda}$ = $(1-\lambda)$.

We have found that this approximation is quite accurate over a wide range of uniform workloads. Figure 14 compares the approximation in Equation 3 with simulation results for both i.i.d. Bernoulli arrivals and for an on-off arrival process modulated by a 2-state Markov-chain.

6 The *i*SLIP Algorithm with Multiple Iterations

Until now, we have only considered the operation of *i*SLIP with a single iteration. We now examine how the algorithm must change when multiple iterations are performed.

With more than one iteration, the iterative *i*SLIP algorithm improves the size of the match: each iteration attempts to add connections not made by earlier iterations. Not surprisingly, we find that the performance

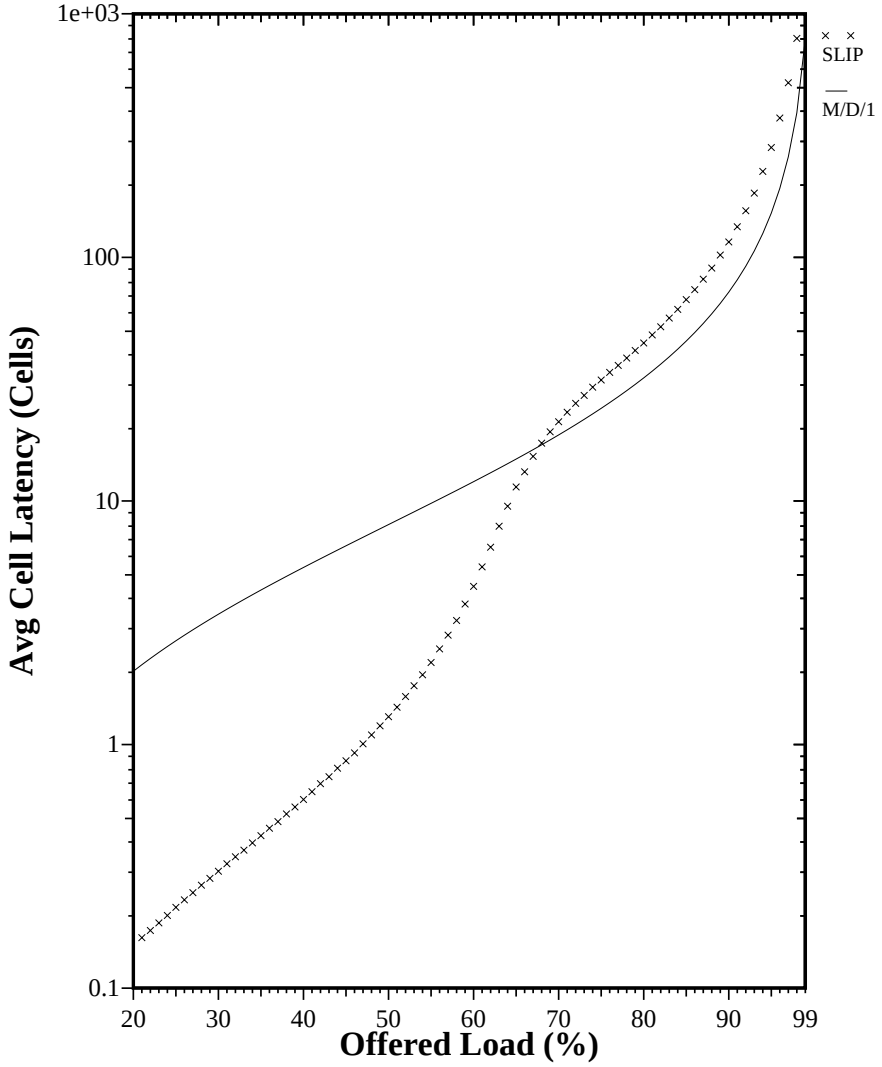


FIGURE 13 Comparison of average latency for the *i*SLIP algorithm and an M/D/1 queue. The switch is 16x16 and, for the *i*SLIP algorithm, arrivals are uniform i.i.d. Bernoulli arrivals.

5.1 Convergence to Time-Division Multiplexing Under Heavy Load

Under heavy load, *i*SLIP will behave similarly to an M/D/1 queue with arrival rates $\frac{\lambda}{N}$ and deterministic service time N cell times. So, under a heavy load of Bernoulli arrivals the delay will be approximated by Equation 2.

To see how close *i*SLIP becomes to time-division multiplexing under heavy load, Figure 13 compares the average latency for both *i*SLIP and an M/D/1 queue (Equation 2). Above an offered load of approximately 70%, *i*SLIP behaves very similarly to the M/D/1 queue, but with a higher latency. This is because the service policy is

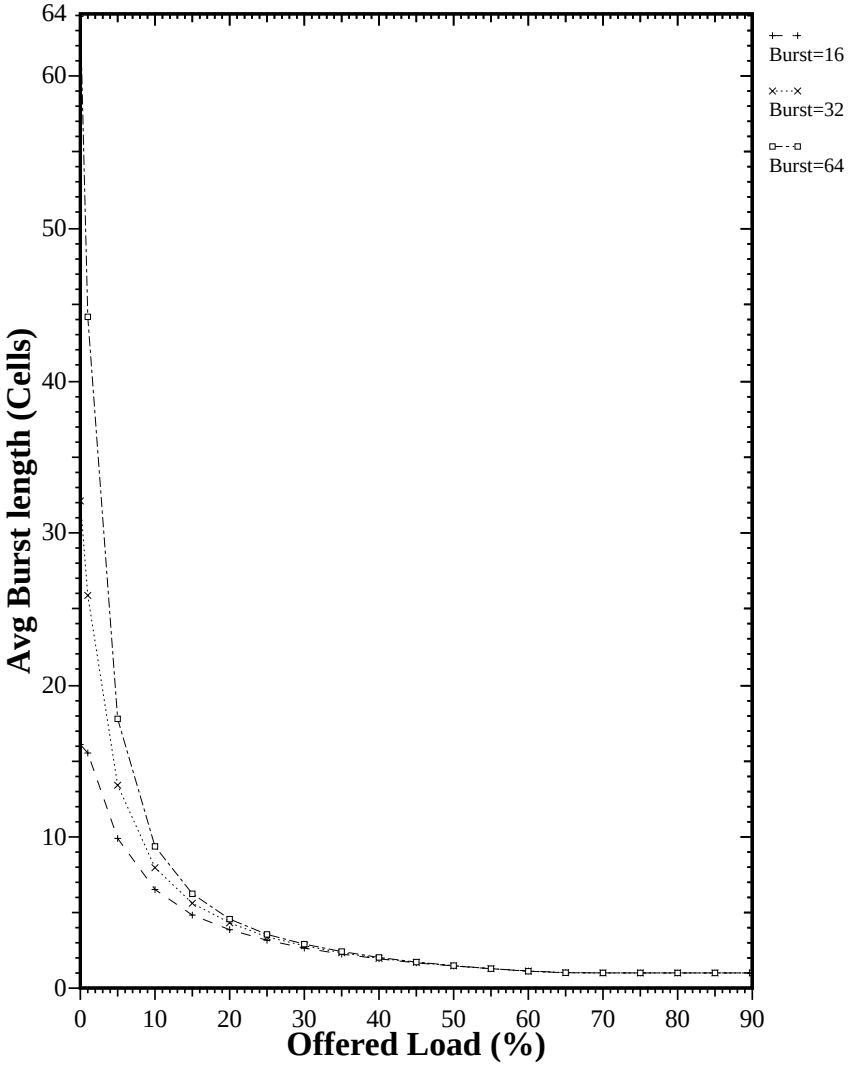


FIGURE 12 Average burst length at switch output as a function of offered load. The arrivals are on-off processes modulated by a 2-state DTMC. Results are for a 16x16 switch using the *i*SLIP scheduling algorithm.

dently and that arriving cells are successfully scheduled with very low delay. At the other extreme, when the switch becomes uniformly backlogged, we can see that desynchronization will lead the arbiters to find an efficient time division multiplexing scheme and operate without contention. But when the traffic is non-uniform, or when the offered load is at neither extreme, the interaction between the arbiters becomes difficult to describe. The problem lies in the evolution and interdependence of the state of each arbiter and their dependence on arriving traffic.

So, for the *i*SLIP switch under a heavy load of Bernoulli arrivals the delay will be approximately

$$d = \frac{\lambda N}{2(1-\lambda)} \tag{2}$$

which is proportional to N.

4.4 Burstiness Reduction:

Intuitively, if a switch decreases the average burst length of traffic that it forwards, then we can expect it to improve the performance of its downstream neighbor. We can expect any scheduling policy that uses round-robin arbiters to be burst-reducing[1] this is also the case for *i*SLIP.

*i*SLIP is a deterministic algorithm, serving each connection in strict rotation. We therefore expect that bursts of cells at different inputs contending for the same output will become interleaved and the burstiness will be reduced. This is indeed the case, as shown in Figure 12. The graph shows the average burst length at the switch output as a function of offered load. Arrivals are on-off processes modulated by a 2-state Markov chain with average burst lengths of 16, 32 and 64 cells.

Our results indicate that *i*SLIP reduces the average burst length, and will tend to be more burst-reducing as the offered load increases. This is because the probability of switching between multiple connections increases as the utilization increases. When the offered load is low, arriving bursts do not encounter output contention and the burst of cells is passed unmodified. As the load increases, the contention increases and bursts are interleaved at the output. In fact, if the offered load exceeds approximately 70%, the average burst length drops to exactly one cell. This indicates that the output arbiters have become desynchronized and are operating as time-division multiplexers, serving each input in turn.

5 Analysis of *i*SLIP Performance

In general, it is difficult to accurately analyze the performance of a *i*SLIP switch, even for the simplest traffic models. Under uniform load and either very low or very high offered load we can readily approximate and understand the way in which *i*SLIP operates. When arrivals are infrequent we can assume that the arbiters act indepen-

1.There are many definitions of burstiness, for example the coefficient of variation [36], burstiness curves [20], maximum burst length [10], or effective bandwidth [21]. In this section, we use the same measure of burstiness that we use when generating traffic: the average burst length. We define a burst of cells at the output of a switch as the number of consecutive cells that entered the switch at the same input.

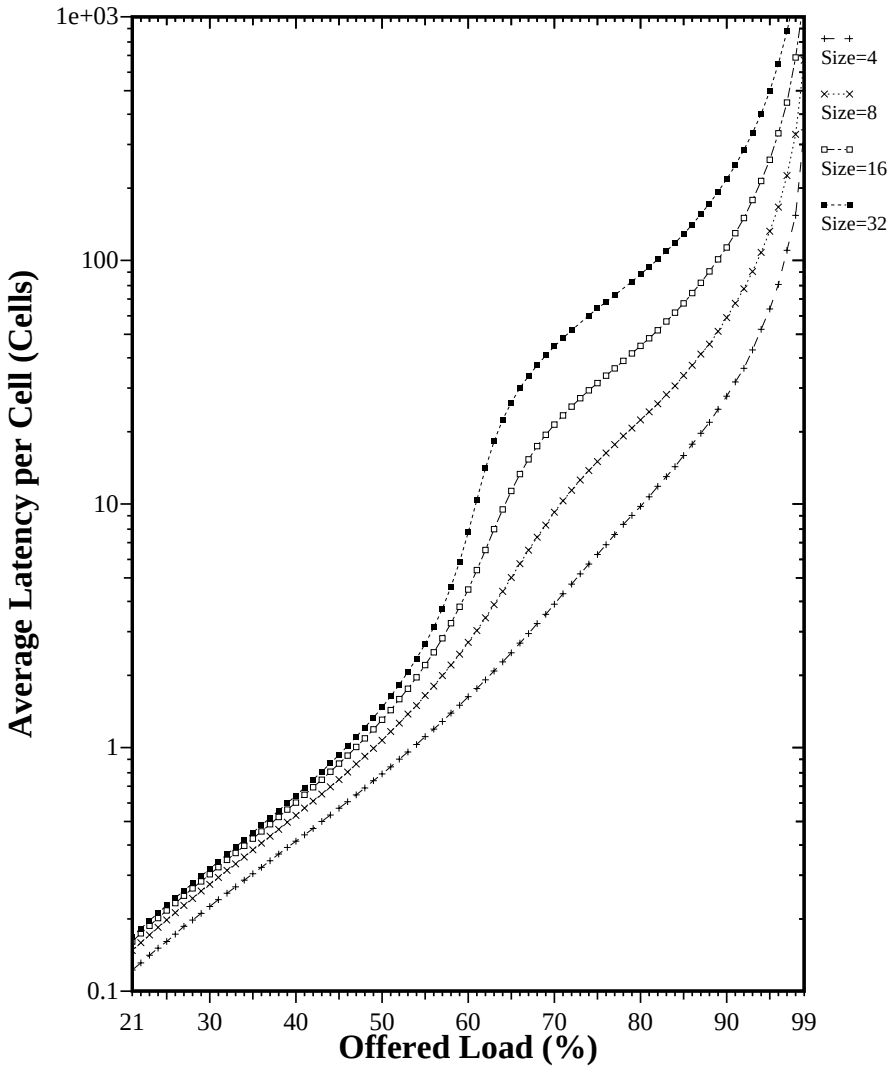


FIGURE 11 The performance of *i*SLIP as function of switch size. Uniform i.i.d. Bernoulli arrivals.

FIFO once every *N* cycles and the queues will behave similarly to an M/D/1 queue with arrival rates $\frac{\lambda}{N}$ and deterministic service time N cell times. For an M/G/1 queue with random service times *S*, arrival rate λ and service rate μ the queueing delay is given by

$$d = \frac{\lambda E(S^2)}{2\left(1 - \frac{\lambda}{\mu}\right)} . \quad (1)$$

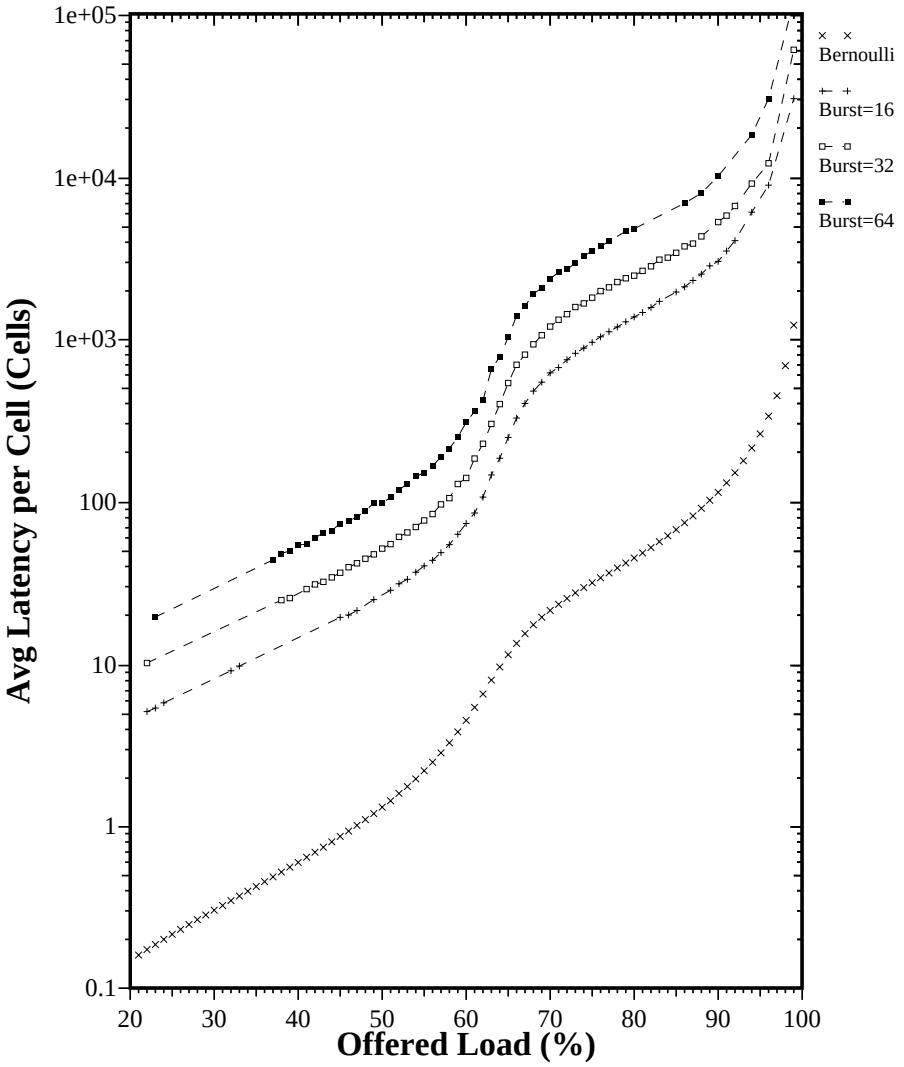


FIGURE 10 The performance of *i*SLIP under 2-state Markov-modulated Bernoulli arrivals. All cells within a burst are sent to the same output. Destinations of bursts are uniformly distributed over all outputs.

degree of synchronization of the arbiters. Under low load, arriving cells find the arbiters in random positions and *i*SLIP performs in a similar manner to the single iteration version of PIM. The probability that the cell is scheduled to be transmitted immediately is proportional to the probability that no other cell is waiting to be routed to the same output. Ignoring the (small) queueing delay under low offered load, the number of contending cells for each output is approximately $\lambda\left(1-\left(\frac{N-1}{N}\right)^{N-1}\right)$ which converges with increasing N to $\lambda\left(1-\frac{1}{e}\right)$.[1] Hence, for constant small λ, the queueing delay converges to a constant as N increases. Under heavy load, the algorithm serves each

1.Note that the convergence is quite fast, and holds approximately even for small N. For example, $1-\left(\frac{N-1}{N}\right)^{N-1}$ equals 0.6073 when $N = 8$, 0.6202 when $N = 16$ and 0.63 when N is infinite.

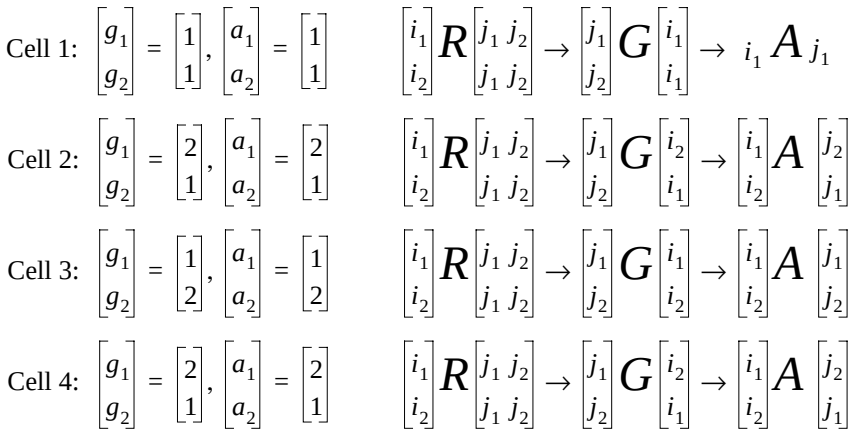


FIGURE 9 Illustration of 100% throughput for *i*SLIP caused by desynchronization of output arbiters. Note that pointers [g_i] become desynchronized at the end of Cell 1 and stay desynchronized, leading to an alternating cycle of 2 cell times and a maximum throughput of 100%.

shows the performance of *i*SLIP under this arrival process for a 16×16 switch, comparing it with the performance under uniform i.i.d. Bernoulli arrivals. The burst length indicated in the graph represents the average length of each busy period. As we would expect, the increased burst size leads to a higher queueing delay. In fact, the average latency is *proportional* to the expected burst length. With bursty arrivals the performance of an input-queued switch becomes more and more like an output-queued switch under the save arrival conditions [9]. This similarity indicates that the performance for bursty traffic is not heavily influenced by the queueing policy, or service discipline. Burstiness tends to concentrate the conflicts on outputs rather than inputs: each burst contains cells destined for the same output and each input will be dominated by a single burst at a time, reducing input contention. As a result, the performance becomes limited by output contention, which is present in both input and output queued switches.

4.3 As a Function of Switch Size:

Figure 11 shows the average latency imposed by a *i*SLIP scheduler as a function of offered load for switches with 4, 8, 16 and 32 ports. As we might expect, the performance degrades with the number of ports.

But the performance degrades differently under low and heavy loads. For a fixed low offered load, the queueing delay converges to a constant value. However, for a fixed heavy offered load the increase in queueing delay is *proportional* to *N*. The reason for these different characteristics under low and heavy load lies once again in the

ority at that output (input). If input i successfully connects to output j, both a_i and g_j are updated and the connection from input i to output j becomes the lowest priority connection in the next cell time.

Property 2. No connection is starved. This is because an input will continue to request an output until it is successful. The output will serve at most $N-1$ other inputs first, waiting at most N cell times to be accepted by each input. Therefore, a requesting input is always served in less than N^2 cell times.

Property 3. Under heavy load, all queues with a common output have the same throughput. This is a consequence of Property 2: the output pointer moves to each requesting input in a fixed order, thus providing each with the same throughput.

But most importantly, this small change prevents the output arbiters from moving in lock-step leading to a large improvement in performance.

4 Simulated Performance of *i*SLIP

4.1 With Benign Bernoulli Arrivals:

Figure 5 shows the performance improvement of *i*SLIP over RRM. Under low load, *i*SLIP's performance is almost identical to RRM and FIFO; arriving cells usually find empty input queues, and on average there are only a small number of inputs requesting a given output. As the load increases, the number of synchronized arbiters decreases (see Figure 8), leading to a large sized match. In other words, as the load increases, we can expect the pointers to move away from each, making it more likely that a large match will be found quickly in the next cell time. In fact, under uniform 100% offered load the *i*SLIP arbiters adapt to a time-division multiplexing scheme, providing a perfect match and 100% throughput. Figure 9 is an example for a 2×2 switch showing how under heavy traffic the arbiters adapt to an efficient time-division multiplexing schedule.

4.2 With Bursty Arrivals:

Real network traffic is highly correlated from cell to cell and so in practice, cells tend to arrive in bursts, corresponding perhaps to a packet that has been segmented or to a packetized video frame. Many ways of modeling bursts in network traffic have been proposed [11], [15], [3], [22]. Leland *et al.* [32] have demonstrated that measured network traffic is bursty at every level making it important to understand the performance of switches in the presence of bursty traffic.

We illustrate the effect of burstiness on *i*SLIP using an on-off arrival process modulated by a 2-state Markov-chain. The source alternately produces a burst of full cells (all with the same destination) followed by an idle period of empty cells. The bursts and idle periods contain a geometrically distributed number of cells. Figure 10

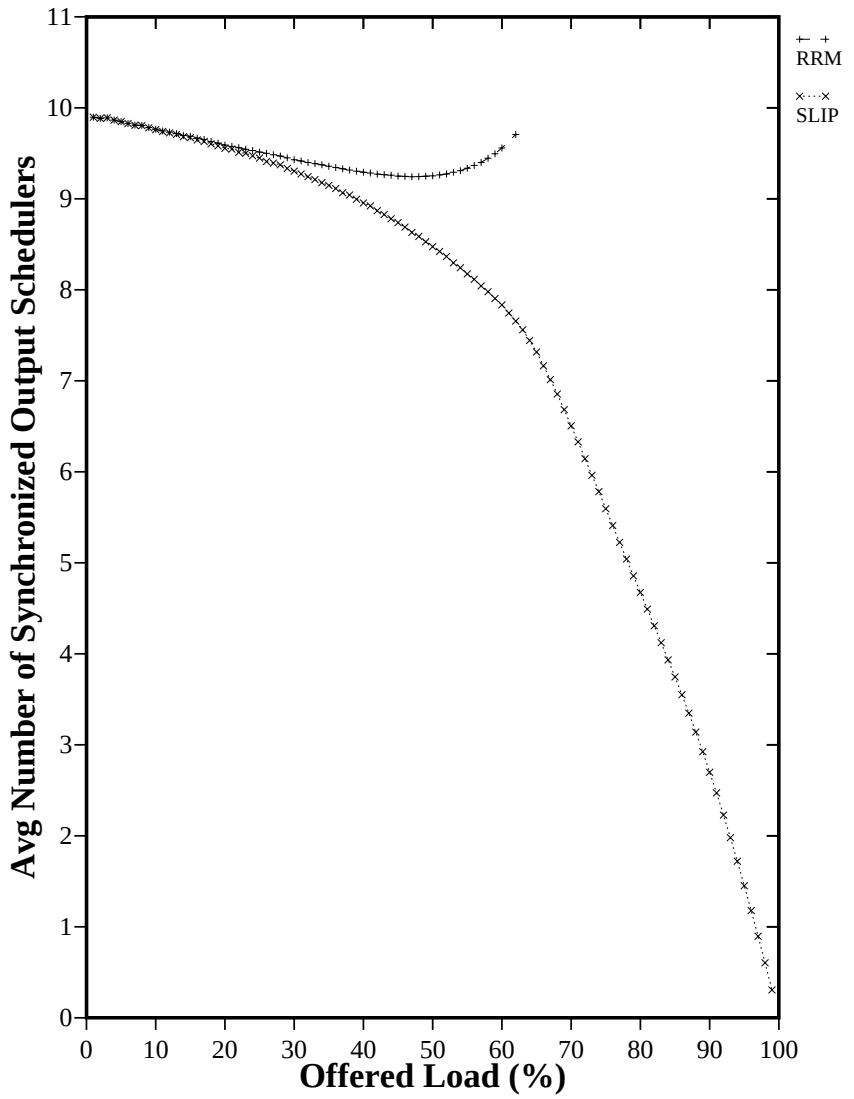


FIGURE 8 Synchronization of output arbiters for RRM and *i*SLIP for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch.

Step 2. *Grant*. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. *The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the granted input if and only if the grant is accepted in Step 3.*

This small change to the algorithm leads to the following properties of *i*SLIP with one iteration:

Property 1. Lowest priority is given to the most recently made connection. This is because when the arbiters move their pointers, the most recently granted (accepted) input (output) becomes the lowest pri-

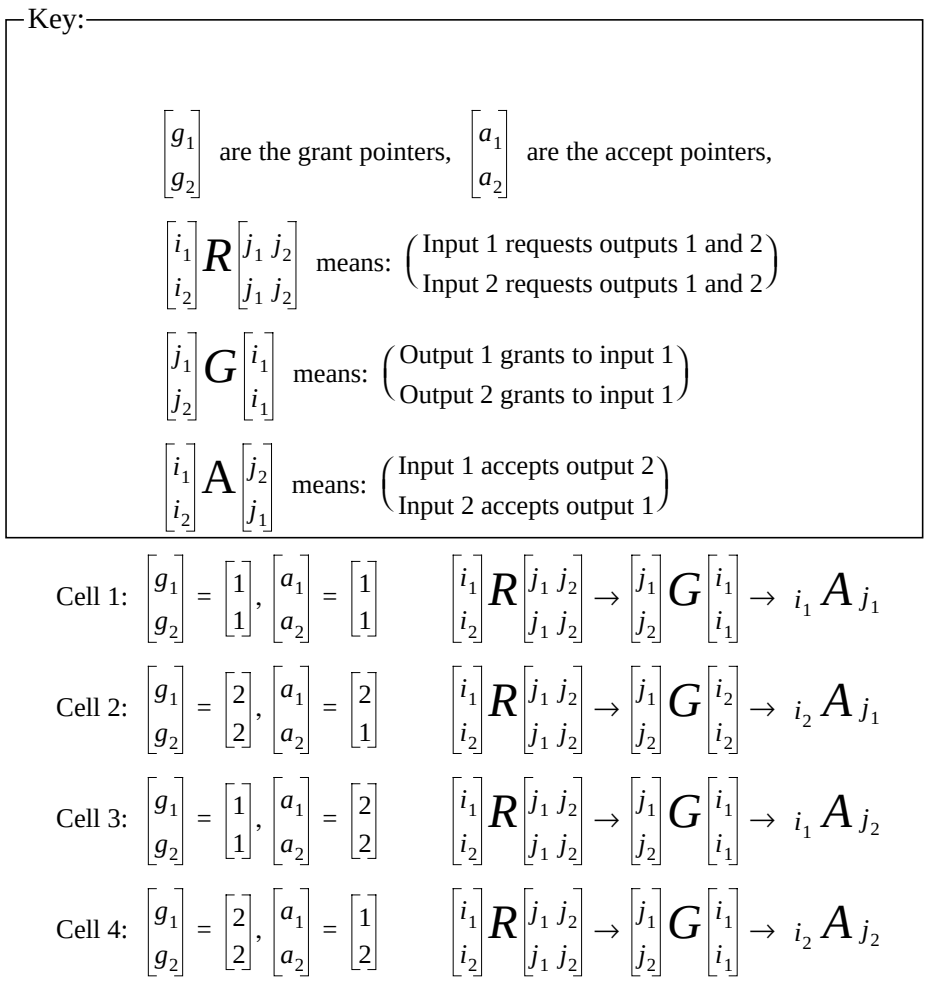


FIGURE 7 Illustration of low throughput for RRM caused by synchronization of output arbiters. Note that pointers [g_i] stay synchronized, leading to a maximum throughput of just 50%.

3 The *i*SLIP Algorithm

The *i*SLIP algorithm improves upon RRM by reducing the synchronization of the output arbiters. *i*SLIP achieves this by not moving the grant pointers unless the grant is accepted. *i*SLIP is identical to RRM except for a condition placed on updating the grant pointers. The *Grant* step of RRM is changed to:

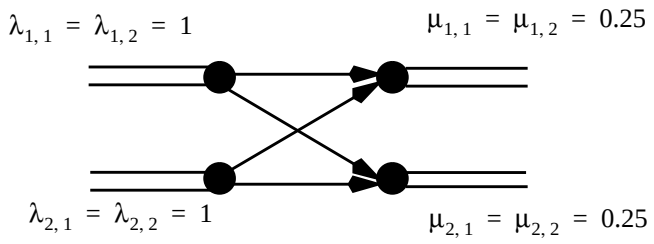


FIGURE 6 2x2 switch with RRM algorithm under heavy load. In the example of Figure 7, synchronization of output arbiters leads to a throughput of just 50%.

The reason for the poor performance of RRM lies in the rules for updating the pointers at the output arbiters. We illustrate this with an example, shown in Figure 6. Both inputs 1 and 2 are under heavy load and receive a new cell for both outputs during every cell time. But because the output schedulers move in lock-step, only one input is served during each cell time. The sequence of requests, grants, and accepts for four consecutive cell times are shown in Figure 7. Note that the grant pointers change in lock-step: in cell time 1 both point to input 1 and during cell time 2 both point to input 2 *etc*. This synchronization phenomenon leads to a maximum throughput of just 50% for this traffic pattern.

Synchronization of the grant pointers also limits performance with random arrival patterns. Figure 8 shows the number of synchronized output arbiters as a function of offered load. The graph plots the number of non-unique g_i's, i.e. the number of output arbiters that clash with another arbiter. Under low offered load, cells arriving for output j will find g_j in a random position, equally likely to grant to any input. The probability that $g_j \neq g_k$ for all $k \neq j$ is $\left(\frac{N-1}{N}\right)^{N-1}$ which for $N = 16$ implies that the expected number of arbiters with the same highest-priority value is 9.9. This agrees well with the simulation result for RRM in Figure 8. As the offered load increases, synchronized output arbiters tend to move in lock-step and the degree of synchronization changes only slightly.

1.The probability that an input will remain ungranted is $\left(\frac{N-1}{N}\right)^{N}$, hence as N increases, the throughput tends to $1-\frac{1}{e} \approx 63\%$

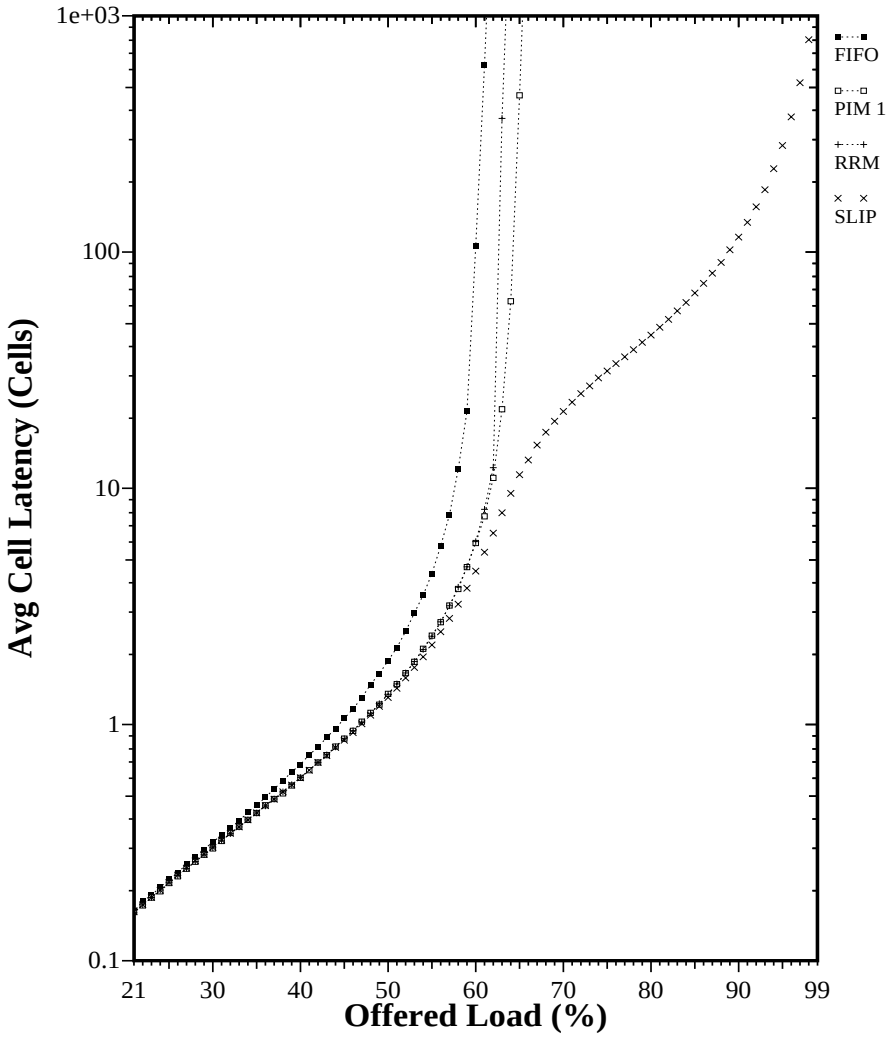


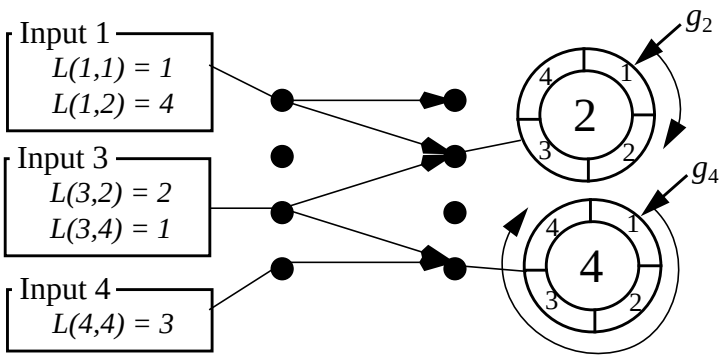
FIGURE 5 Performance of RRM and *i*SLIP compared with PIM for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch. The graph shows the average delay per cell, measured in cell times, between arriving at the input buffers and departing from the switch.

Step 2. *Grant*. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo *N*) to one location beyond the granted input.

Step 3. *Accept*. If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo *N*) to one location beyond the accepted output.

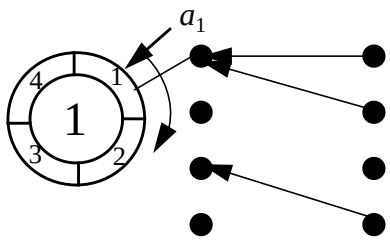
2.2 Performance of RRM for Bernoulli Arrivals

As an introduction to the performance of the RRM algorithm, Figure 5 shows the average delay as a function of offered load for uniform i.i.d. Bernoulli arrivals. For an offered load of just 63% RRM becomes unstable.[1]

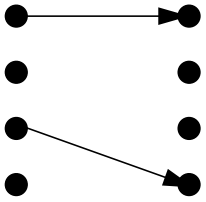


a) Step 1: *Request*. Each input makes a request to each output for which it has a cell.

Step 2: *Grant*. Each output selects the next requesting input at or after the pointer in the round-robin schedule. Arbiters are shown here for outputs 2 and 4. Inputs 1 and 3 both requested output 2. Since $g_2 = 1$ output 2 grants to input 1. g_2 and g_4 are updated to favor the input after the one that is granted.



b) Step 3: *Accept*. Each input selects at most one output. The arbiter for input 1 is shown. Since $a_1 = 1$ input 1 accepts output 1. a_1 is updated to point to output 2.



c) When the arbitration has completed, a matching of size two has been found. Note that this is less than the maximum sized matching of three.

FIGURE 4 Example of the three steps of the RRM matching algorithm.

algorithm, like PIM, consists of three steps. As shown in Figure 4, for an $N \times N$ switch each round-robin schedule contains *N* ordered elements. The three steps of arbitration are:

Step 1. *Request*. Each input sends a request to every output for which it has a queued cell.

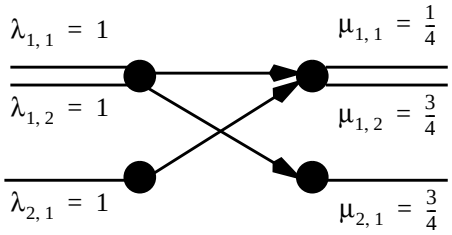


FIGURE 3 Example of unfairness for PIM under heavy, oversubscribed load with more than one iterations. Because of the random and independent selection by the arbiters, output 1 will grant to each input with probability 1/2, yet input 1 will only accept output 1 a quarter of the time. This leads to different rates at each output.

This is because the probability that an input will remain ungranted is $\left(\frac{N-1}{N}\right)^N$, hence as N increases, the throughput tends to $1-\frac{1}{e} \approx 63\%$. Although the algorithm will often converge to a good match after several iterations, the time to converge may affect the rate at which the switch can operate. We would prefer an algorithm that performs well with just a single iteration.

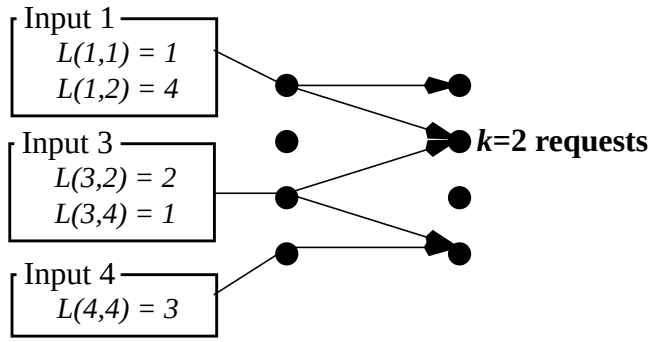
2 The *i*SLIP Algorithm with a Single Iteration

In this section we describe and evaluate the *i*SLIP algorithm. This section concentrates on the behavior of *i*SLIP with just a single iteration per cell time. Later, we will consider *i*SLIP with multiple iterations.

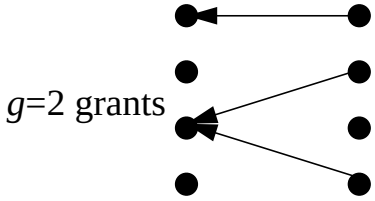
The *i*SLIP algorithm uses rotating priority ("round-robin") arbitration to schedule each active input and output in turn. The main characteristic of *i*SLIP is its simplicity: it is readily implemented in hardware and can operate at high speed. We find that the performance of *i*SLIP for uniform traffic is high; for uniform i.i.d. Bernoulli arrivals, *i*SLIP with a single iteration can achieve 100% throughput. This is the result of a phenomenon that we encounter repeatedly: the arbiters in *i*SLIP have a tendency to *desynchronize* with respect to one another.

2.1 Basic Round-Robin Matching Algorithm

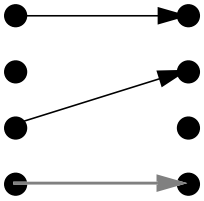
*i*SLIP is a variation of simple basic round-robin matching algorithm (RRM). RRM is perhaps the simplest and most obvious form of iterative round-robin scheduling algorithms, comprising a two-dimensional array of round-robin arbiters: cells are scheduled by round-robin arbiters at each output, and at each input. As we shall see, RRM does not perform well; but it helps us to understand how *i*SLIP performs, so we start here with a description of RRM. RRM potentially overcomes two problems in PIM: *complexity* and *unfairness*. Implemented as priority encoders, the round-robin arbiters are much simpler and can perform faster than random arbiters. The rotating priority aids the algorithm in assigning bandwidth equally and more fairly among requesting connections. The RRM



a) Step 1: *Request*. Each input makes a request to each output for which it has a cell. This is shown here as a graph with all weights, $w_{i,j} = 1$.



b) Step 2: *Grant*. Each output selects an input uniformly among those that requested it. In this example, inputs 1 and 3 both requested output 2. Output 2 chose to grant to input 3.



c) Step 3: *Accept*. Each input selects an output uniformly among those that granted to it. In this example, outputs 2 and 4 both granted to input 3. Input 3 chose to accept output 2.

FIGURE 2 An example of the three steps that make up one iteration of the PIM scheduling algorithm [2]. In this example, the first iteration does not match input 4 to output 4, even though it does not conflict with other connections. This connection would be made in the second iteration.

service. Third, it means that no memory or state is used to keep track of how recently a connection was made in the past. At the beginning of each cell time, the match begins over, independently of the matches that were made in previous cell times. Not only does this simplify our understanding of the algorithm, but it also makes analysis of the performance straightforward: there is no time-varying state to consider, except for the occupancy of the input queues.

But using randomness comes with its problems. First, it is difficult and expensive to implement at high speed: each arbiter must make a random selection among the members of a time-varying set. Second, when the switch is oversubscribed, PIM can lead to unfairness between connections. An extreme example of unfairness for a 2×2 switch when the inputs are oversubscribed is shown in Figure 3. We will see examples later for which PIM and some other algorithms are unfair when no input or output is oversubscribed. Finally, PIM does not perform well for a single iteration: it limits the throughput to approximately 63%, only slightly higher than for a FIFO switch.

aggregate bandwidth of 0.5Tb/s [26]. *i*SLIP is based on the Parallel Iterative Matching algorithm (PIM) [2], and so to understand its operation, we start by describing PIM. Then, in Section 2 we describe *i*SLIP and its performance. We then consider some small modifications to *i*SLIP for various applications, and finally consider its implementation complexity.

1.2 Parallel Iterative Matching

Parallel Iterative Matching (PIM) was developed by DEC Systems Research Center for the 16-port, 1Gb/s AN2 switch [2].[1] Because it forms the basis of the *i*SLIP algorithm described later, we will describe the scheme in detail and consider some of its performance characteristics.

PIM uses *randomness* to avoid starvation, and to reduce the number of iterations needed to converge on a maximal sized match. A maximal sized match (a type of on-line match) is one that adds connections incrementally, without removing connections made earlier in the matching process. In general, a maximal match is smaller than a maximum sized match, but is much simpler to implement. PIM attempts to quickly converge on a conflict-free maximal match in multiple iterations, where each iteration consists of three steps. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. The three steps of each iteration operate in parallel on each output and input and are shown in Figure 2. The steps are:

Step 1. *Request*. Each unmatched input sends a request to every output for which it has a queued cell.

Step 2. *Grant*. If an unmatched output receives any requests, it grants to one by randomly selecting a request uniformly over all requests.

Step 3. *Accept*. If an input receives a grant, it accepts one by selecting an output randomly among those that granted to this output.

By considering only unmatched inputs and outputs, each iteration only considers connections not made by earlier iterations.

Note that the independent output arbiters *randomly* select a request among contending requests. This has three effects: first the authors in [2] show that each iteration will match or eliminate on average at least $\frac{3}{4}$ of the remaining possible connections and thus the algorithm will converge to a maximal match, on average, in $O(\log N)$ iterations. Second, it ensures that all requests will eventually be granted, ensuring that no input queue is starved of

1.This switch was commercialized as the Gigaswitch/ATM.

tite matching on a graph with N vertices [2][25][35]; a procedure too complex to implement and run quickly in hardware. For example, the algorithms described in [25] and [28] that achieve 100% throughput, use maximum weight bipartite matching algorithms [35] which have a running time complexity of $O(N^3 \log N)$.

1.1 Maximum Size Matching

Most scheduling algorithms described previously are heuristic algorithms that approximate a maximum *size*[1] matching [1][2][5][8][18][30][36]. These algorithms attempt to maximize the number of connections made in each cell time, and hence maximize the instantaneous allocation of bandwidth. The maximum size matching for a bipartite graph can be found by solving an equivalent network flow problem [35] and we shall call the algorithm that does this *maxsize*. There exist many maximum size bipartite matching algorithms, and the most efficient currently known converges in $O(n^{5/2})$ time [12].[2] The problem with this algorithm is that although it is guaranteed to find a maximum match, for our application it is too complex to implement in hardware and takes too long to complete.

One question worth asking is: Does the *maxsize* algorithm maximize the throughput of an input-queued switch? The answer is no: *maxsize* can cause some queues to be starved of service indefinitely. Furthermore, when the traffic is non-uniform, *maxsize* cannot sustain very high throughput [25]. This is because it does not consider the backlog of cells in the VOQs, or the time that cells have been waiting in line to be served.

For practical high performance systems, we desire algorithms with the following properties:

- *High Throughput* — An algorithm that keeps the backlog low in the VOQs. Ideally, the algorithm will sustain an offered load up to 100% on each input and output.
- *Starvation Free* — The algorithm should not allow a non-empty VOQ to remain unserved indefinitely.
- *Fast* — To achieve the highest bandwidth switch, it is important that the scheduling algorithm does not become the performance bottleneck. The algorithm should therefore find a match as quickly as possible.
- *Simple to implement* — If the algorithm is to be fast in practice, it must be implemented in special-purpose hardware; preferably within a single chip.

The *i*SLIP algorithm presented in this paper is designed to meet these goals, and is currently implemented in a 16-port commercial IP router with an aggregate bandwidth of 50Gb/s [6], and a 32-port prototype switch with an

1. In some literature, the maximum *size* matching is called the maximum *cardinality* matching or just the maximum bipartite matching.
2. This algorithm is equivalent to Dinic's algorithm [9].

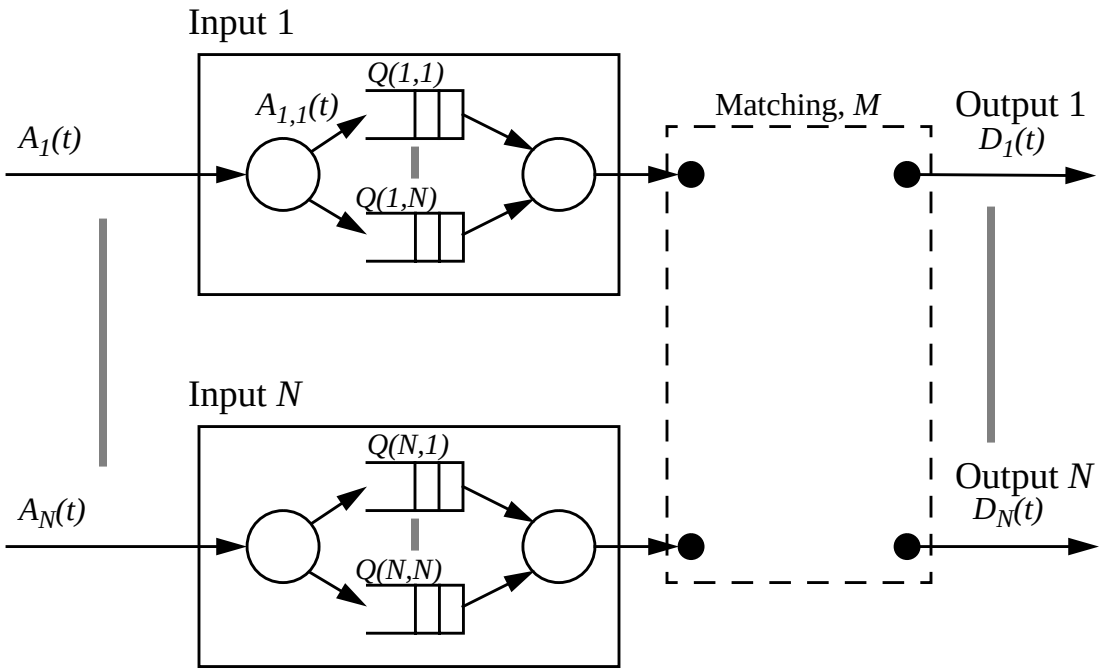


FIGURE 1 An input-queued switch with "virtual output queueing". Note that head of line blocking is eliminated by using a separate queue for each output at each input.

suggested for reducing HOL blocking, for example by considering the first K cells in the FIFO queue, where $K>1$ [8][13][17]. Although these schemes can improve throughput, they are sensitive to traffic arrival patterns and may perform no better than regular FIFO queueing when the traffic is bursty. But HOL blocking can be eliminated by using a simple buffering strategy at each input port. Rather than maintain a single FIFO queue for all cells, each input maintains a separate queue for each output as shown in Figure 1. This scheme is called "virtual output queueing" (VOQ) and was first introduced by Tamir *et al*. in [34]. HOL blocking is eliminated because cells only queue behind cells that are destined to the same output: no cell can be held up by a cell ahead of it that is destined to a different output. When VOQs are used, it has been shown possible to increase the throughput of an input-queued switch from 58.6% to 100% for both uniform and non-uniform traffic [25][28]. Crossbar switches that use VOQs have been employed in a number of studies [1][14][19][23][34], research prototypes [26][31][33], and commercial products [2][6]. For the rest of this paper, we will be considering crossbar switches that use VOQs.

When we use a crossbar switch, we require a scheduling algorithm that configures the fabric during each cell time, and decides which inputs will be connected to which outputs; this determines which of the N^2 VOQs are served in each cell time. At the beginning of each cell time, a scheduler examines the contents of the N^2 input queues and determines a conflict-free match, M, between inputs and outputs. This is equivalent to finding a bipar-

hardware. Our work was motivated by the design of two such systems: the Cisco 12000 GSR, a 50Gb/s IP router, and the *Tiny Tera:* a 0.5Tb/s MPLS switch [7].

Before using a crossbar switch as a switching fabric, it is important to consider some of the potential drawbacks; we consider three here. First, the implementation complexity of an N-port crossbar switch increases with N^2, making crossbars impractical for systems with a very large number of ports. Fortunately, the majority of high performance switches and routers today have only a relatively small number of ports (usually between 8 and 32). This is because the highest performance devices are used at aggregation points where port density is low.[1] Our work is therefore focussed on systems with low port density. A second potential drawback of crossbar switches is that they make it difficult to provide guaranteed qualities of service. This is because cells arriving to the switch must contend for access to the fabric with cells at both the input and the output. The time at which they leave the input queues and enter the crossbar switching fabric is dependent on other traffic in the system making it difficult to control when a cell will depart. There are two common ways to mitigate this problem. One is to schedule the transfer of cells from inputs to outputs in a similar manner to that used in a time-slot interchanger, providing peak bandwidth allocation for reserved flows. This method has been implemented in at least two commercial switches and routers.[2] The second approach is to employ "speedup" in which the core of the switch runs faster than the connected lines. Simulation and analytical results indicate that with a small speedup, a switch will deliver cells quickly to their outgoing port, apparently independent of contending traffic [27][37][38][39][40][41]. While these techniques are of growing importance, we restrict our focus in this paper to the efficient and fast scheduling of *best-effort* traffic.

A third potential drawback of crossbar switches is that they (usually) employ input-queues. When a cell arrives, it is placed in an input queue where it waits its turn to be transferred across the crossbar fabric. There is a popular perception that input-queued switches suffer from *inherently* low performance due to head of line (HOL) blocking. HOL blocking arises when the input buffer is arranged as a single FIFO queue: a cell destined to an output that is free may be held up in line behind a cell that is waiting for an output that is busy. Even with benign traffic, it is well-known that HOL can limit thoughput to just $2-\sqrt{2} \approx 58.6\%$ [16]. Many techniques have been

1. Some people believe that this situation will change in the future, and that switches and routers with large aggregate bandwidths will support hundreds or even thousands of ports. If these systems become real, then crossbar switches — and the techniques that follow in this paper — may not be suitable. However, the techniques described here will be suitable for a few years hence.

2. A peak-rate allocation method was supported by the DEC AN2 Gigaswitch/ATM [2] and the Cisco Systems LS2020 ATM Switch.

The *i*SLIP Scheduling Algorithm for Input-Queued Switches

Nick McKeown
Department of Electrical Engineering
Stanford University, Stanford, CA 94305-9030
nickm@stanford.edu

Abstract

An increasing number of high performance IP routers, LAN switches and ATM switches use a switched backplane based on a crossbar switch. Most often, these systems use input queues to hold packets waiting to traverse the switching fabric. It is well-known that if simple FIFO input queues are used to hold packets then, even under benign conditions,head-of-line (HOL) blocking limits the achievable bandwidth to approximately 58.6% of the maximum. HOL blocking can be overcome by the use of virtual output queueing — which is described in this paper. A scheduling algorithm is used to configure the crossbar switch, deciding the order in which packets will be served. Recent results have shown that with a suitable scheduling algorithm 100% throughput can be achieved. In this paper, we present a scheduling algorithm called iSLIP. An iterative, round-robin algorithm, iSLIP can achieve 100% throughput for uniform traffic, yet is simple to implement in hardware. Iterative and non-iterative versions of the algorithms are presented, along with modified versions for prioritized traffic. Simulation results are presented to indicate the performance of iSLIP under benign and bursty traffic conditions. Prototype and commercial implementations of iSLIP exist ing systems with aggregate bandwidths ranging from 50-500Gb/s. When the traffic is non-uniform, iSLIP quickly adapts to a fair scheduling policy that is guaranteed never to starve an input queue. Finally, we describe the implementation complexity of iSLIP. Based on a two-dimensional array of priority encoders, single-chip schedulers have been built supporting up to 32-ports, and making approximately 100million scheduling decisions per second.

1 Introduction

In an attempt to take advantage of ATM's cell switching capacity, there has recently been a merging of ATM switches and IP routers [29][32]. This idea is already being carried one step further, with cell-switches forming the core, or backplane, of high performance IP routers [26][31][6][4]. Each of these high speed switches and routers is built around a crossbar switch that is configured using a centralized scheduler, and each uses a fixed size cell as a transfer unit. Variable length packets are segmented as they arrive, transferred across the central switching fabric, then reassembled again into packets before they depart. A crossbar switch is used because it is simple to implement, and is non-blocking: it allows multiple cells to be transferred across the fabric simultaneously, alleviating the congestion found on a conventional shared backplane. In this paper, we describe an algorithm that is designed to configure a crossbar switch using a single-chip, centralized scheduler. The algorithm presented here attempts to achieve high throughput for best-effort unicast traffic, and is designed to be simple to implement in