A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches.

Adisak Mekkittikul Nick McKeown
Computer Systems Laboratory
Stanford University, Stanford, CA 94305-9030
{adisak, nickm}@stanford.edu

Abstract

Input queueing is becoming increasingly used for high-bandwidth switches and routers. In previous work, it was proved that it is possible to achieve 100% throughput for input-queued switches using a combination of virtual output queueing and a scheduling algorithm called LQF. However, this is only a theoretical result: LQF is too complex to implement in hardware. In this paper we introduce a new algorithm called Longest Port First (LPF), which is designed to overcome the complexity problems of LQF, and can be implemented in hardware at high speed. By giving preferential service based on queue lengths, we prove that LPF can achieve 100% throughput.

1 Introduction

Traditionally, switches and routers have been most often designed as a collection of line-cards connected to a single shared bus. Packets waiting to be transmitted on outgoing links are stored in a centralized, shared pool of memory. If the aggregate bandwidths of the bus and memory are high enough, the system is able to keep all of the outgoing links continuously busy, making the system highly efficient. Furthermore, the system is able to control packet departure times and hence provides guaranteed qualities-of-service (QoS) [3][15][20][21]. However, switch and router designers are finding that the continued growth in bandwidth is making it increasingly difficult to design a shared bus and centralized memory that run fast enough. The data rate of a shared bus is limited by electrical considerations, such as the loading on the bus, and reflections from connectors. And the data rate of a centralized shared memory is limited because it requires buffer memories that run N times faster than the line rate, where N is the number of switch ports.

This work was funded by a fellowship from National Semiconductor and also by Texas Instruments, Cisco Systems, the Alfred P. Sloan Foundation and a Robert N. Noyce faculty fellowship.

Increasingly, a passive shared bus is being replaced by an active non-blocking switch fabric — most often a crossbar switch. Each line card is connected by a dedicated point-to-point link to the central switch fabric, and therefore has fewer electrical limitations due to loading and reflections. More importantly, each connection to the switch need run only as fast as the line rate, rather than at the aggregate bandwidth of the switch. Centralized shared memory is also being replaced—by separate queues at each input of the switching fabric. Input queues need only run at the line rate, and therefore allow a faster overall system to be built [6][11].

The very fastest switches and routers usually transfer packets across the switching fabric in fixed size units, that we shall refer to as "cells." Variable length packets are segmented into cells upon arrival, transferred across the switch fabric and then reassembled again before they depart. At the beginning of each cell time, a (usually centralized) *scheduler* selects a configuration for the switching fabric and then transfers cells from inputs to outputs. Using fixed sized cells simplifies the switch design, and makes it easier for the scheduler to configure the switch fabric for high throughput.

But systems that use input queues have two potential problems: low throughput due to head-of-line (HOL) blocking and the difficulty of controlling cell delay. In this paper, we focus on the first problem: achieving high throughput.

It is well known that if an input-queued switch employs a single FIFO queue at each input, HOL blocking limits the throughput to just 58.6% of the maximum [7]. But HOL blocking can be eliminated entirely using a queueing technique known as *virtual output queueing* (VOQ) in which each input maintains a separate queue for each output [1][10][12][13][17]. It has been shown that with a suitable centralized scheduling algorithm, the throughput can be increased from 58.6% to 100% [12].

Unfortunately, the algorithms known to-date (LQF [12] and OCF [13]) are too complex to implement in hardware, and are therefore unsuitable for switches operating at high speed. Instead, most switches and routers use a much simpler scheduling algorithm to configure the switch fabric [1][10][18]. Typically, a configuration is selected in an attempt to maximize the number of connections made during each cell time. Such an algorithm is called a maximum size bipartite matching algorithm, and is found to perform well when the arriving traffic is *uniformly* distributed over all the switch outputs.

But real traffic is not uniform: traffic tends to be focused on a relatively small number of active ports. And unfortunately, a maximum size matching algorithm is known to perform poorly when traffic is non-uniform [12]. The algorithm performs poorly in two (albeit related) ways: increased buffer overflows, and reduced throughput. Increased overflows occur because a maximum size matching algorithm does not consider queue lengths when deciding which input queues to service. When traffic is non-uniform, the occupancies of the various input queues can differ greatly, and queues with heavy traffic can overflow while ones with light traffic remain empty most of the time. The reason for reduced throughput is a little more complex. For a given number of cells in the system, if the traffic is non-uniform, the cells are concentrated on a relatively small number of VOQs. This reduces the number of configurations available to the scheduler, and therefore reduces the size of the maximum size match. If instead the traffic was uniform, the cells in the system would be distributed uniformly over a relatively large number of VOQs, making available a larger number of configurations for the scheduler to choose from.

In earlier work [12][13], it was found that LQF (longest queue first) can achieve 100% for both uniform and non-uniform traffic by considering the occupancies of the queues. LQF gives preferential service to long queues by using a maximum *weight* matching algorithm, where each weight is set to the corresponding queue length. But LQF is very difficult to implement in hardware at high speed. First of all, it takes too long to run—the most efficient algorithm known to-date has a running-time complexity $O(N^3 \log N)$. Second, an implementation requires a large number of multi-bit comparators to perform many weight comparisons in parallel. Attempts to implement LQF (and even heuristic approximations [10]) have been limited by the design of a single-chip scheduler that: (i) has fast enough comparators, (ii) can support a sufficient number of comparators, and (iii) can interconnect them in a rich enough pattern.

Motivated by the desire to overcome the impracticalities of LQF, yet achieve its high performance, we propose a new algorithm: LPF (longest port first). With LPF our goal is to combine the benefits of a maximum size matching algorithm, with those of a maximum weight algorithm, while lending itself to simple implementation in hardware. LPF effectively finds the *set* of maximum size matches, and from among this set chooses the match with the largest total weight. In LPF each weight is a function of queue lengths (we shall see later that the weights in LPF are not exactly equal to the queue lengths, but are similar). This enables LPF to take advantage of both the high *instantaneous* throughput of a maximum size matching algorithm, and the ability of a maximum weight matching algorithm to achieve high throughput, and a small number of overflows even when the arriving traffic is non-uniform. We find that LPF—like LQF—can achieve 100% throughput for both uniform and non-uniform traffic.

LPF has a running-time complexity of $O(N^{2.5})$; lower than LQF. Furthermore, the comparators that limit the performance of LQF are removed from the critical path of the LPF algorithm. In fact, the heart of the LPF algorithm uses a slightly modified maximum *size* matching algorithm, for which there are a variety of existing, heuristic approximations [1][9][10][17].

The paper is organized as follows. In Section 2, we provide some definitions. In Section 3, we describe LPF and its properties before presenting our performance analysis.

2 Our Switch Model

We follow the general definitions used in [12]. Figure 1 shows an $M \times N$ input-queued switch consisting of M input and N output ports, a non-blocking switching fabric and a scheduler. To eliminate head-of-line (HOL) blocking, each input maintains N FIFO virtual output queues, one for each output. $Q_{i,j}$ denotes the VOQ at input i containing cells destined to output j. Arrivals are fixed size packets or cells, allowing us to split time into discrete cell times, or *slots*. During any given slot, there is at most one arrival to and departure from each input, and similarly for each output. $A_{i,j}(n)$ is the arrival process of cells to input i destined to output j at rate $\lambda_{i,j}$. Consequently, $A_i(n)$ is the aggregate process of all arrivals to input i at rate $\lambda_i = \sum_{j=1}^{N} \lambda_{i,j}$.

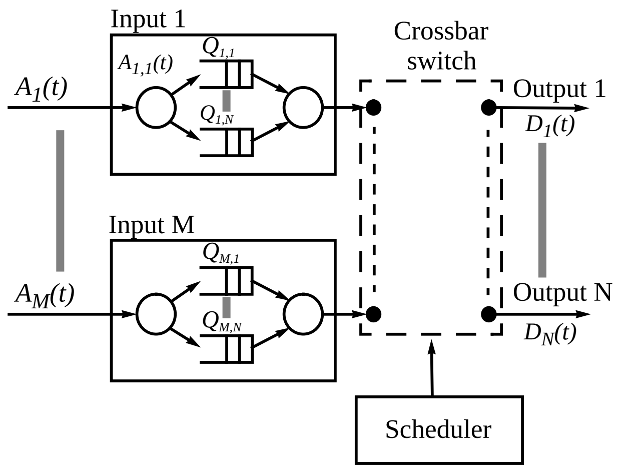

Figure 1: A Simple Model of VOQ Switches.

Definition 1: *An arrival process is said to be admissible when no input or output is oversubscribed, i.e., when* $\sum_{i=1}^{M} \lambda_{i,j} < 1$, $\sum_{j=1}^{N} \lambda_{i,j} < 1$, $\lambda_{i,j} \geq 0$.

Definition 2: *The traffic is uniform if all arrival processes have the same arrival rate, and if the destinations of cells are uniformly distributed over all outputs. Otherwise the traffic is non-uniform.*

The scheduler determines which inputs and outputs are connected during each slot. The scheduling problem can be viewed as a bipartite graph matching problem [2][19], an example of which is shown in Figure 2. Each input makes a request to every output for which it has cells queued. An edge in the graph represents a request from $Q_{i,j}$ with weight $w_{i,j}(n)$ (denoted in Figure 2 as $w_{i,j}$).

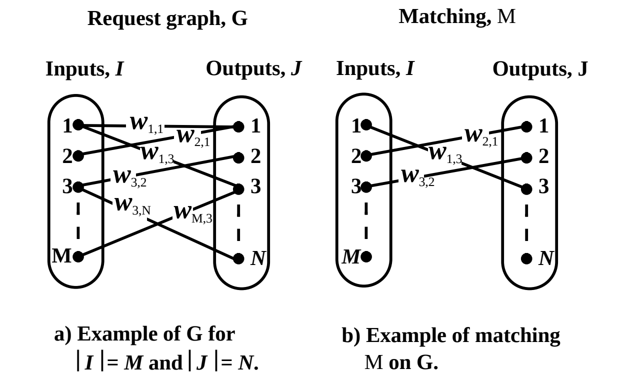

Figure 2: A request graph and a matching graph of an $M \times N$ switch. Define G = [V,E] as an undirected graph connecting the set of vertices V with the set of edges E. The edge connecting vertices i, $1 \leq i \leq M$ and j, $1 \leq j \leq N$ has an associated weight denoted $w_{i,j}$. Graph G is bipartite if the set of inputs $I = \{i: 1 \leq i \leq M\}$ and outputs $J = \{i: 1 \leq j \leq N\}$ partition V such that every edge has one end in I and one end in J. Matching M on G is any subset of E such that no two edges in M have a common vertex.

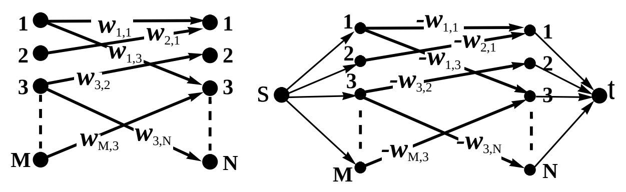

Figure 3: Transformation of a request graph into a flow network. (a) A weighted request graph. (b) The corresponding flow network, G, whose all edges are of unity capacity. A source s and a target t are added. The cost of every edge from s and to t is set to zero. The cost of all other edges are equal to the negated value of the corresponding weight.

Let $S_{i,j}(n)$ be a service indicator such that $\sum_{i=1}^{M} S_{i,j}(n) \leq 1$ and $\sum_{j=1}^{N} S_{i,j}(n) \leq 1$; a value of one indicates that input i is matched to output j, i.e., $Q_{i,j}$ is allowed to forward one cell to its output.

Definition 3: *A maximum size match is one that maximizes* $\sum_{i,j} S_{i,j}(n)$*, i.e., the number of connections.*

Definition 4: *A maximum weight match is one that maximizes* $\sum_{i,j} S_{i,j}(n) w_{i,j}(n)$*, i.e., the total weight.*

Alternatively, a bipartite graph matching problem can be easily solved and understood by transforming it into a flow network [2][19], as illustrated in Figure 3.

3 The LPF Algorithm

Although in practice LPF can be thought of as a special maximum size matching algorithm, in theory it is easier to consider LPF as a maximum weight matching algorithm. Each LPF request weight, $w_{i,j}(n)$, for a request from input i to output j is defined as follows:

$$w_{i,j}(n) = \begin{cases} R_i(n) + C_j(n), & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \quad (1)$$

Modified Edmonds-Karp algorithm

```
1 for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3         $f[v, u] = 0$
4 while LPFS finds a path $p$ from $s$ to $t$ in the residual network $G_f$
5     for each edge $(u, v)$ in $p$
6         do if $f[u, v] = 0$ then $f[u, v] = c[u, v]$
7             else $f[v, u] \leftarrow 0$
8         $f[u, v] = -f[v, u]$
```

Figure 4: Modified Edmonds-Karp algorithm [2]. G is a flow network or graph constructed as described in Figure 3. $E[G]$ is the set of all edges in G; u or v is a vertex in G representing an input or output; (u, v) is an edge from u to v; f is the total flow through the network; $f[u, v]$ denotes a flow from u to v. G_f is a residual network [2] [19], also called a residual graph. LPFS is a largest unmatched port first search.

LPFS(G)

```
1 for each vertex $u \in V[G]$
2     do $color[u] \leftarrow$ white
3         $\pi[u] \leftarrow$ nil
4 LPFS-Visit(t)
```

LPFS-Visit(u)

```
1 $color[u] \leftarrow$ gray
2 for each $v \in Adjacent[u]$, starting the largest to the smallest.
3     do if $color[v] =$ white
4         then $\pi[v] \leftarrow u$
5         LPFS-Visit(v)
6 $color[u] =$ black
```

Figure 5: A largest-unmatched-port first search (LPFS). First, LPFS builds a tree with t as its root. Initially every input and output is colored white — undiscovered, then is grayed when it is discovered, and finally is blackened when it is finished. $\pi[v]$ is the predecessor of v. From the tree, an augmenting path from s to t which must go through an unmatched input can be found by walking the predecessor list which begins at a selected unmatched input.

where $L_{i,j}(n)$ is the occupancy of $Q_{i,j}$ at slot n, $R_i(n) = \sum_j^N L_{i,j}(n)$ and $C_j(n) = \sum_i^N L_{i,j}(n)$.

$R_i(n)$, which we call the input occupancy, is the total number of cells that are currently waiting at input i to be forwarded to their respective outputs. Similarly, $C_j(n)$, the output occupancy, is the total number of cells at all inputs waiting to be forwarded to output j. Together, the sum of the input and output occupancies represents the work load or congestion that a cell faces as it competes for transmission to its output. We call this sum the *port occupancy*; LPF favors queues with high port occupancy.

Property 1: *The total weight of an LPF match is equal to the occupancy sum of all matched inputs and outputs, i.e,* $\sum_{i,j} S_{i,j}(n) w_{i,j}(n) = \sum_{i \in I} R_i + \sum_{j \in J} C_j$, *where I and J are the set of matched inputs and matched outputs respectively.*

We now show that LPF is a special case of a maximum size matching algorithm.

Theorem 1: *LPF finds a match that is both maximum size and maximum weight.*

Proof of Main Theorem: see Appendix A. ❐

Since an LPF match is a maximum size match, we can use a maximum size matching algorithm to find an LPF match. But we need to make sure that among all possible maximum size matches we choose one with the largest total weight.

3.1 Finding an LPF Match Using a Maximum Size Matching Algorithm

Existing maximum size matching algorithms cannot be used to implement LPF because they are unable to select the maximum size match with the largest weight. A simple modification is called for. First, in order to keep the algorithm free of complex magnitude comparisons, all inputs and outputs are *pre-ordered* according to their LPF weights prior to running the maximum size matching algorithm. Then we use a modified Edmonds-Karp maximum size matching algorithm [2][19] to find the LPF match (see Figure 4). A breadth-first search (BFS) in the Edmonds-Karp algorithm is replaced by a largest-unmatched-port first search (LPFS) described in Figure 5. LPFS enables the modified algorithm to search for a maximum weight match while performing path augmentation [19] to find a maximum size match. As a result, line 2 of the LPFS-Visit does not involve any magnitude comparison. It is proved in [14] that the modified algorithm finds an LPF match.

Iterative LPF algorithm

```
Step 1.
1   Sort inputs&outputs based on their occupancies
2   Reorder requests according to their input and output
    occupancies

Step 2. Maximal size matching
1   for each output from largest->smallest
2       for each input from largest->smallest
3           if (there is a request) and (both input and output
              unmatched)
4             then match them
```

Figure 6: An iterative LPF algorithm. First, the algorithm builds a sorted list of all inputs and outputs based on their occupancies. Then, starting from the largest output and input, the algorithm finds a maximal size match.

Theorem 2: *The maximum size match found by the modified Edmonds-Karp algorithm is also a maximum weight match with weights as defined in Equation 1.*

Proof of Main Theorem: see reference [14]. ❐

3.2 A Practical Approximation to LPF

LPF can be adapted to run at higher speed using simple heuristic approximations. Shown in Figure 6 is an iterative algorithm called *i*LPF that approximates LPF. All weight processing is done in step 1 prior to the iterative steps. The second step consists of a double for-loop used to find a *maximal* size match. Since the requests have already been ordered in the first step, the maximal size matching in the second step does not need to compare request weights. Figure 7 shows the schematic of a hardware implementation of *i*LPF. Our exploratory design work suggests that the second step can be implemented using simple hardware; for a 32×32 switch, our synthesized design can make a scheduling decision in just 10ns using a commercial $0.25\mu m$ CMOS ASIC technology. The first step, which requires simple integer arithmetic, can also run in 10ns, allowing the switch to run at a line rate of 20 Gb/s.[1]

3.3 Stability

We now prove that LPF can achieve 100% throughput for all traffic patterns with independent arrivals, using the notion of *stability* [8]. We define a switch to be stable for a particular arrival process if the expected length of the input queues does not grow without bound, i.e.,

$$E\left[\sum_{i,j} L_{i,j}(n)\right] < \infty, \forall n . \qquad (2)$$

Definition 5: *A switch can achieve 100% throughput if it is stable for all independent and admissible arrivals.*

Theorem 3: *The LPF algorithm is stable for all admissible independent arrival processes.*

Proof of Main Theorem: see Appendix B. ❐

3.4 Stability With a Finite Pipeline Delay

Because the modified maximum size matching algorithm requires the input and outputs to be pre-ordered, LPF and *i*LPF need sorting networks to sort all inputs and outputs. Due to the relatively high complexity of the sorting networks, they could dominate the running time of the algorithm. Alternatively, we can pipeline the design to reduce its running time; the sorting networks can operate in one slot, and the maximum size matching algorithm in the next. This means that the maximum size matching algorithm is operating on weights that are now one slot out of date — it is possible for the algorithm to favor the

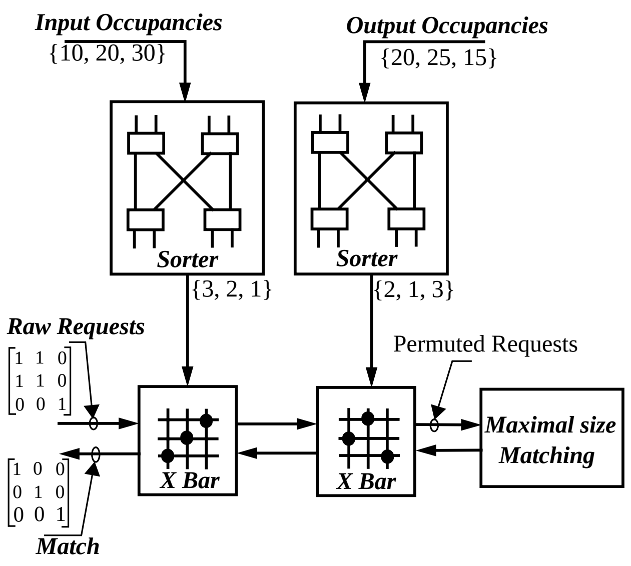

Figure 7: A block diagram of *i*LPF. Referring to the algorithm in Figure 6, inputs and outputs are pre-sorted by the two sorter networks. Raw requests (requests with weights removed) is given in a matrix form. Request reordering is done by the two crossbars which are configured by the sorting results. The maximal size matching block, which implements the double for-loop, finds a maximal size match that approximates an LPF match. The match needs to be permuted back to its natural order.

1. Calculated based on the size of an ATM cell.

wrong queues, or worse still, schedule a queue that is now empty. However, because of the speed benefits of pipelining, we consider here its effect on throughput.

A k slot pipeline delay is equivalent to non-pipelined LPF but with k slot old weights, $w_{i,j}(n-k)$. Hence, it finds the match that maximizes $\sum_{i,j} S_{i,j}(n)w_{i,j}(n-k)$. Perhaps surprisingly, we can verify the following:

Theorem 4: *Using k slot old weights, the LPF algorithm is stable for all admissible independent arrival processes,* $0 < k < \infty$.

Proof of Main Theorem: see Appendix B. ❐

4 Conclusion

Input-queued non-blocking switches offer much higher aggregate bandwidth than systems based on shared buses and centralized shared memory. While VOQs make it theoretically possible for an input-queued switch to achieve high throughput, most existing scheduling algorithms yield low throughput or are too complex to run at high speed. Our new scheduling algorithm, LPF, is both practical, and can achieve 100% throughput for all traffic with independent arrivals. Because LPF uses a maximum *size* matching algorithm, it leads to a fast, iterative, heuristic algorithm called *i*LPF that is simple to implement in hardware. Initial investigation suggests that *i*LPF can configure a 32×32 switch in 10ns using today's ASIC technology. Furthermore, we find that with pipelining, LPF can be operated even faster without loss of average throughput.

5 References

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

[2] Cormen, T.; Leiserson C.E.; Rivest R.L. "Introduction to algorithms," The MIT Press, Cambridge, Massachusetts, March 1990.

[3] Demers, A., et al. "Analysis and simulation of a fair queueing algorithm," *Internetworking: Research and Experience*, vol.1, no.1, Sept. 1990, pp. 3-26.

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

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

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

[7] Karol, M.; Hluchyj, M.; and Morgan, S. "Input versus output queueing on a space division switch," *IEEE Trans. Communications*, vol.35, no. 12, 1987, pp.1347-1356.

[8] Kumar, P.R.; Meyn, S.P.; "Stability of queueing networks and scheduling policies," *IEEE Transactions on Automatic Control*, vol.40, no.2, Feb. 1995, pp. 251-260.

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

[10] McKeown, N. "Scheduling Algorithms for Input-Queued Cell Switches," PhD Thesis. University of California at Berkeley, 1995.

[11] McKeown, N.; Izzard M.; Mekkittikul, A.; Ellersick W.; Horowitz M. "The Tiny Tera: a packet switch core," *IEEE Micro*, vol 17, no. 1, Jan/Feb. 1997, pp. 27-33.

[12] McKeown, N.; Anantharam, V.; and Walrand, J. "Achieving 100% throughput in an input-queued switch," *Proceedings of INFOCOM*,1996, pp. 296-302.

[13] Mekkittikul, A.; McKeown, N "A Starvation-free Algorithm for Achieving 100% Throughput in an Input-Queued Switch," *Proceedings of ICCCN'96*, October, 1996, pp. 226-231.

[14] Mekkittikul, A.; McKeown, N "Scheduling VOQ switches under non-uniform traffic," CSL Technical report, CSL-TR-97-747, Stanford University, 1997.

[15] Parekh, A.K.; Gallager, R. "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case," *IEEE/ACM Transactions on Networking*, vol.2, no.2, April 1994, pp. 137-50.

[16] Partridge, C.; et al. "A fifty gigabit per second IP router," accepted for publication in *IEEE/ACM Transactions on Networking*.

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

[18] Tamir, Y. et al. "Symmetric crossbar arbiters for VLSI communication switches" *IEEE Transactions on Parallel and Distributed Systems*, vol.4, no.1, Jan 1993., pp. 13-27.

[19] Tarjan, R.E. "Data structures and network algorithms," *Society for Industrial and Applied Mathematics*, Pennsylvania, Nov 1983.

[20] Zhang, H. "Service disciplines for guaranteed performance service in packet-switching networks" *Proceedings of the IEEE*, vol.83, no.10, Oct, 1995, pp. 1374-96.

[21] Zhang, L. "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Transactions on Computer Systems*, vol.9, no.2, May 1991, pp. 101-24.

Appendix A: An LPF Match Property

Theorem 1: *LPF finds a match that is both maximum size and maximum weight.*

Proof of Main Theorem: We prove the theorem by contradiction. Let M be a maximum weight match found by LPF that is not a maximum size match; i.e. there exists a maximum size match $M' \neq M$ such that $|M'| > |M|$.[1]

Using the Ford-Fulkerson method [2], without removing any previously matched input or output from M, a larger match M' can be found by augmenting a flow on the corresponding flow network. As a result, the matched sets in M' contain all the inputs and outputs of the matched sets in M and at least one additional input and one additional output. According to Property 1, M' is clearly a larger weight match than M. Therefore, M cannot be the match found by the LPF because it is not a maximum weight match. Therefore, M must be both a maximum size and maximum weight match. ❐

Appendix B: LPF Stability

B.1 Definitions

We use the following, additional definitions:

1. The rate matrix of the stationary arrival processes:

$$\Lambda \equiv [\lambda_{i,j}], \quad \text{where: } \sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0$$

and associated rate vector:

$$\underline{\lambda} \equiv (\lambda_{1,1}, \ldots, \lambda_{1,N}, \ldots, \lambda_{N,1}, \ldots, \lambda_{N,N})^T .$$

2. The arrival matrix, representing the sequence of arrivals into each queue:

$$\mathbf{A}(n) \equiv [A_{i,j}(n)] ,$$

where:

$$A_{i,j}(n) \equiv \begin{cases} 1 & \text{if an arrival occurs at } Q_{i,j} \text{ at slot } n \\ 0 & \text{else,} \end{cases}$$

and associated arrival vector:

$$\underline{A}(n) \equiv (A_{1,1}(n), \ldots, A_{1,N}(n), \ldots, A_{N,N}(n))^T .$$

3. The service matrix, indicating which queues are served at time n:

$$\mathbf{S}(n) \equiv [S_{i,j}(n)] \text{, where:}$$

$$S_{i,j}(n) \equiv \begin{cases} 1 & \text{if } Q_{i,j} \text{ is served at time } n \\ 0 & \text{else,} \end{cases}$$

and $\mathbf{S}(n) \in \mathbf{S}$, the set of service matrices.

Note that: $\sum_{i=1}^{N} S_{i,j}(n) = \sum_{j=1}^{N} S_{i,j}(n) = 1$, and $\mathbf{S}(n) \in \mathbf{S}$ is a *permutation matrix*. We define the associated service vector:

$$\underline{S}(n) \equiv (S_{1,1}(n), \ldots, S_{1,N}(n), \ldots, S_{N,N}(n))^T .$$

4. Hence we can define the next-state occupancy vector:

$$\underline{L}(n+1) \equiv \underline{L}(n) - \underline{S}(n) + \underline{A}(n) ; \qquad (3)$$

5. A positive-definite, symmetric transformation matrix, T. For an $N \times N$ switch, T is an $N^2 \times N^2$ matrix whose elements are defined as follows:

$$T_{i,j} \equiv \begin{cases} 2, \text{ when } i = j \\ 1, \text{ when} \left\lfloor \frac{i}{N} \right\rfloor = \left\lfloor \frac{j}{N} \right\rfloor \\ 1, \text{ when } i \bmod N = j \bmod N \\ 0, \text{ otherwise.} \end{cases}$$

B.2 Stability of LPF without a Pipeline Delay

B.2.1 Main Theorem

Theorem 3: *The LPF algorithm is stable for all admissible independent arrival processes.*

B.2.2 Proof

Consider the quadratic Lyapunov function, $V(\underline{L}(n)) = \underline{L}(n) T \underline{L}(n)$ and the LPF request vector $\underline{L}'(n)$ whose elements are a function of queue occupancies defined as:

$$L'_{i,j}(n) = w_{i,j}(n) , \qquad (4)$$

where $w_{i,j}(n)$ is as defined in Equation 1. Note that be-

1. The size of $M \equiv |M|$.

cause of our definition of T

$$\underline{L}'(n) = T\underline{L}(n) . \qquad (5)$$

The following lemmas lead to our proof of the main theorem. Due to a limited space, we refer readers to [14] for the detailed proof of Lemmas 1, 2 and 3.

Lemma 1: $E[\underline{L}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n)] \le 0$ *for* $\sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0$ *, where* $\underline{S}^*(n) = \arg[\max(\underline{L}'^T(n)\underline{S}(n))]$.

Lemma 1 is true because a match found by LPF is a maximum weight match whose weights are as defined in Equation 1 and Equation 4. Furthermore, considering the definition of admissibility, we can refine Lemma 1 to:

Lemma 2: $E[\underline{L}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n)] \le -2\beta \sum_{i,j} L_{i,j}(n)$ *for any* $\underline{\lambda} \le (1-\beta)\underline{\lambda}_m$ *, where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j} \underline{\lambda}_{m_{i,j}} = N$ *and* $0 < \beta < 1$,.

Lemma 2 leads us to the following result that under LPF there exists a single-step negative drift in the Lyapunov function.

Lemma 3: *Under the LPF algorithm, there exists* $\varepsilon > 0$ *such that*

$$\mathrm{E}[\underline{L}^T(n+1)T\underline{L}(n+1) - \underline{L}^T(n)T\underline{L}(n) | \underline{L}(n)] \le -\varepsilon \sum_{i,j} L_{i,j}(n) + 2N^2$$

$\forall \underline{\lambda} \le (1-\beta)\underline{\lambda}_m, 0 < \beta < 1$ *, where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j} \underline{\lambda}_{m_{i,j}} = N$.

Now, we are ready to prove the main theorem.

Proof of Main Theorem:

Lemma 3 shows that there exists a *quadratic Lyapunov function,* $V(\underline{L}(n)) = \underline{L}(n)T\underline{L}(n)$, such that:

$$\mathrm{E}[V(\underline{L}(n+1)) - V(\underline{L}(n)) | \underline{L}(n)] \le -\varepsilon \sum_{i,j} L_{i,j}(n) + 2N^2 . \qquad (6)$$

According to Kumar [8], the above implies that the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{T+1} \sum_{n=0}^{T} \sum_{i,j} E[L_{i,j}(n)] < \infty, \forall N , \qquad (7)$$

and so each queue occupancy is bounded. In particular, if the arrivals are independent, the $\underline{L}(n)$ forms a Markov chain, and Equation 7 guarantees its positive recurrence. ❐

B.3 Stability with a Pipeline Delay

B.3.1 Main Theorem

Theorem 4: *Using weights that are k slots old, the LPF algorithm is stable for all admissible independent arrival processes,* $0 < k < \infty$.

B.3.2 Proof

The proof will use the following lemma.

Lemma 4: $\underline{L}'(n)\underline{S}^*(n) - \underline{L}'(n)\tilde{\underline{S}}(n) \le (N^2 + 3N)k$ *, where* $\underline{S}^*(n)$ *is the optimum service vector if LPF had been given the correct weights, and* $\tilde{\underline{S}}(n)$ *is the service vector selected by LPF using the k-slot old weights.*

We refer to [14] for the detailed proof of this lemma. In brief, Lemma 4 indicates that the consequence of the pipeline delay is limited. There is a finite and constant bound on the difference between the two total weights.

Now, we can use Lemma 4 to prove the main theorem.

Proof of Main Theorem:

Similar to the proof of Theorem 3, by taking the same steps as used in Lemmas 1 to 3, and by using the relationship described by Lemma 4, we can show that there exists a *quadratic Lyapunov function,* $V(\underline{L}(n)) = \underline{L}(n)T\underline{L}(n)$, such that:

$$\mathrm{E}[V(\underline{L}(n+1)) - V(\underline{L}(n)) | \underline{L}(n)] \le -\varepsilon \sum_{i,j} L_{i,j}(n) + 2N^2 + (N^2 + 3N)k . \text{❐} \qquad (8)$$