Switching Using Parallel Input–Output Queued Switches With No Speedup

Saad Mneimneh, Vishal Sharma, *Senior Member, IEEE*, and Kai-Yeung Siu

***Abstract*—We propose an efficient parallel switching architecture that requires no speedup and guarantees bounded delay. Our architecture consists of k input–output-queued switches with first-in-first-out queues, operating at the line speed in parallel under the control of a *single* scheduler, with k being independent of the number N of inputs and outputs. Arriving traffic is demultiplexed (spread) over the k identical switches, switched to the correct output, and multiplexed (combined) before departing from the parallel switch.**

We show that by using an appropriate demultiplexing strategy at the inputs and by applying the *same matching* at each of the k parallel switches during each cell slot, our scheme guarantees a way for cells of a flow to be read in order from the output queues of the switches, thus, eliminating the need for cell resequencing. Further, by allowing the scheduler to examine the state of only the first of the k parallel switches, our scheme also reduces considerably the amount of state information required by the scheduler. The switching algorithms that we develop are based on existing practical switching algorithms for input-queued switches, and have an additional communication complexity that is optimal up to a constant factor.

***Index Terms*—Delay guarantee, parallel switches, speedup, switching.**

I. Introduction

TRADITIONAL output-queued or shared memory architectures are becoming increasingly inadequate to meet high-bandwidth requirements, because having to account for multiple arrivals to the same output requires their switch memories to operate at N times the line speed, where N is the number of inputs. Although input-queued switches provide an attractive alternative since their memory and switch fabrics may operate at only the line speed, they present a challenge for providing quality-of-service (QoS) guarantees comparable to those provided by output-queued switches, and require a sophisticated scheduler or arbiter, making it a critical component of the switch.

Manuscript received March 13, 2001; revised December 11, 2001; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor N. McKeown. This work was supported in part by Tellabs Research Center, Cambridge, MA, and in part by the Networking Research Program, National Science Foundation, under Award 9973015. This paper was presented in part at the IEEE Workshop on High Performance Switching and Routing (HPSR), Dallas, TX, May 2001.

S. Mneimneh was with the Massachusetts Institute of Technology, Cambridge, MA 02139 USA. He is now with the Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275 USA (e-mail: saad@alum.mit.edu).

V. Sharma is with Metanoia Inc., Mountain View, CA 94041 USA (e-mail: v.sharma@ieee.org).

K.-Y. Siu is with the Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: siu@perth.mit.edu).

Digital Object Identifier 10.1109/TNET.2002.803919.

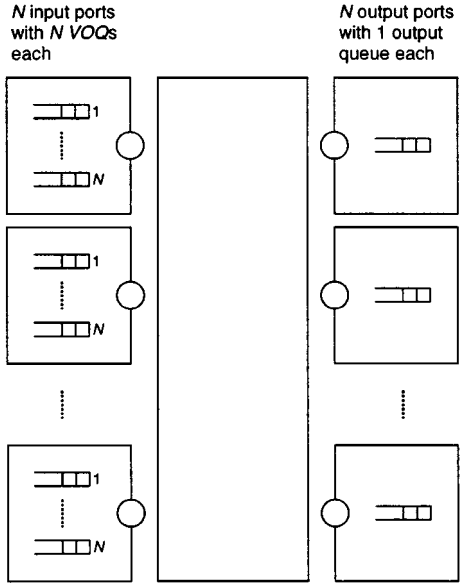

Fig. 1. Input–output-queued switch.

For instance, traditional switching algorithms that achieve 100% throughput in an input-queued switch do not provide delay guarantees, and are based on computing a maximum weighted matching that requires a running time of $O(N^3)$ [10], [11], or $O(N^{2.5})$ [12], making them impractical to implement on high-speed switches. Some recent work has, therefore, focused on asking whether an input-queued switch can be made to emulate an output-queued switch, and has demonstrated that this can be achieved by a combination of a speedup in the fabric (of $2 - (1/N)$) and a special switching algorithm based on computing a stable marriage matching. Such emulation involves substantial bookkeeping and communication overhead at the scheduler, however, and despite its theoretical significance, is not yet practical at high speeds. Most practical switching algorithms for input-queued switches (see, for instance, [3], [9]), therefore, require a speedup of between 2 and 4 to achieve adequate QoS guarantees. This means that the switch fabric and the memory need to operate faster than the line speed by the speedup factor. Note also that an input-queued switch with a speedup will require queues at the output as well, since more than one cell can be forwarded to an output in a single cell slot. Fig. 1 depicts the traditional architecture of an input–output-queued switch with the virtual output queues (VOQs), where all queues are first-in–first-out (FIFO) queues.

We propose to use multiple input–output-queued switches in parallel, allowing each switch to operate at the line speed, so that no speedup is necessary. We show that such an architecture,

combined with efficient switching algorithms, is both feasible and practical, and that operating the switches in parallel incurs only a small additional computational and communication cost. We show how to guarantee a bounded cell delay with this architecture.

The remainder of this paper is organized as follows. In Section II, we motivate the rationale behind the parallel switch approach, while in Section III, we describe the parallel architecture, and outline some of the issues that arise when using switches in parallel. We also describe in Section III the basic idea that motivates our design of a switching algorithm for this parallel switching architecture. In Section IV, we provide a general framework for a switching algorithm that guarantees bounded cell delay. We conclude in Section V.

II. Motivation

As discussed above, most practical switching algorithms require a speedup of at least 2. This poses two nontrivial difficulties in moving toward higher speed switches.

- The memory within the switch must run at a speed faster than that of the external lines. This reduces memory access times, and makes it difficult to build practically usable memories, especially with continuously increasing line speeds.
- With speedup, the time available to obtain a matching (by execution of the switching algorithm) is also reduced. This is particularly problematic for some of the more complex switching algorithms needed to provide guarantees. Specifically, with a speedup of S, a switching algorithm has only $1/S$ time units to compute a matching.

Our approach, therefore, is to eliminate the need for speedup by using input–output-queued switches in parallel. It should be noted that a previous work that addresses the use of parallel switches appears in [6]. Below, we briefly point out some differences between our approach and [6].

- In [6], the authors use parallel output-queued switches, while we use parallel input–output-queued switches, thus, offering a different theoretical framework for the problem.
- The objective in [6] is to emulate output queueing for a switch operating at a high line speed by using a number of output-queued switches operating in parallel at some submultiple of the line speed. Our objective is to provide basic guarantees, such as bounded delay on every cell, without requiring any speedup in the system.
- The algorithm in [6] relies on simulating an output-queued switch in the background, which requires the maintenance of a large amount of state information. Our switching algorithms, on the other hand, are based on existing switching algorithms for an input–output-queued switch that do not require an excessive amount of state information.
- The architecture in [6] naturally requires $2N$ parallel layers (where N is the size of the switch) of output-queued switches to fully eliminate memory speedup in the system. This is because the queue memory of each switch is required to operate at a speed equal to $2RN/k$, where R is the line speed and k is the number of parallel switches. This dependence on N can be removed if input–output-queued switches are used instead (four of them). As a consequence, each input–output-queued switch will then have to emulate an output-queued switch. While such an emulation is possible as demonstrated in [4], it is not yet practical due to the excessive bookkeeping and communication needed between the switching algorithm and the switches. Moreover, the emulation makes use of non-FIFO queues. Nevertheless, the emulation algorithm provided in [4] is practical at low speeds, suggesting that increasing the number of parallel input–output-queued switches renders the algorithm practical. This, however, implies that the number of parallel switches needed has a dependence on the line speed, even if switches operating at the line speed are available. By contrast, to eliminate speedup, our architecture uses a constant number of layers that is independent of N.
- The bandwidth of the architecture in [6] is $2NR$ where R is the line speed. The bandwidth of our architecture is kNR. Therefore, for $k = 2$, which is sufficient to provide delay guarantees, as will be seen later, both architectures have the same bandwidth. A more recent work [7] by the same authors of [6] illustrates an output queueing emulation up to an additive constant factor D using N output queues with no speedup. Hence, they reduce the bandwidth required to NR only. This, however, requires resequencing of cells at the output. But since there is a bound D on the time a cell will be delayed from its output queueing time, resequencing can be eliminated by waiting a time D before delivering any cell at the output. The remaining disadvantage is that $D = 2kN$ where k is the number of switches, and, hence, the delay is $O(N^2)$.

Our main goal is not to emulate output queueing, as was done in [6] and [7]. Rather, it is to obtain an efficient and practical way of achieving basic guarantees, such as bounded delay on every cell, with a constant number of parallel layers, no speedup, and without the need to resequence cells at the output.

III. Parallel Architecture

We use an architecture similar to the architecture described in [6]. The only difference between the architecture presented here and that of [6] is that we use input–output-queued switches while the authors in [6] use output-queued switches. The architecture is depicted in Fig. 2.

The architecture consists of the N input ports having a demultiplexer each, and the N output ports having a multiplexer each. The middle stage consists of k switches in parallel, with each switch being an input–output-queued switch, like the one depicted in Fig. 1. At each input port i, a demultiplexer sends a cell arriving on that input to one of the k parallel switches. Likewise, at every output port j, a multiplexer accesses the output queue for that port (i.e., the jth output queue) in each of the k switches. Since no speedup is to be used, we define a cell slot to be the time needed for a cell to be read from or stored into a queue. Therefore, the switches operate in cell slots where in each cell slot, each switch can forward at most one cell from an input port and at most one cell to an output port. Although we assume that no speedup is being used, the switches of Fig. 2 are input–output-queued switches for the following reason: Since there is no speedup, an output port can deliver at most one cell

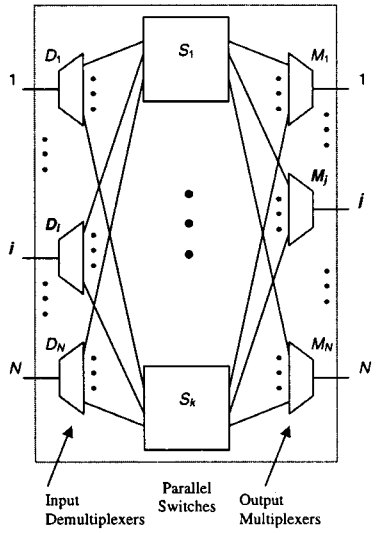

Fig. 2. Parallel switches.

per cell slot; however, multiple cells can be forwarded to that output by multiple switches during a single cell slot. Hence, forwarded cells need to be stored.

During each cell slot, multiple cells may arrive at an input i provided each is destined to a different output j. The actual arrival pattern, of course, depends on the traffic model and on the specific implementation of the demultiplexers (for example, each demultiplexer in Fig. 2 can represent N actual demultiplexers for the different N flows at the input).

To proceed further, we define the following notation:

(i, j)	flow (of cells) from input i to output j.
$C(i, j)$	a cell from input i to output j.
$P(i, j)$	a cell from input i to output j.
VOQ_{ij}^{l}	VOQ_{ij} in switch l.
OQ_{j}^{l}	Output queue j in switch l.

Unless otherwise mentioned, in the proofs that follow, we neither require any synchronization between cell arrivals and the operation of the parallel switches, nor do we require any synchronization between the k switches themselves, except that they all perform a matching by the end of a cell slot. Our problem is to find a switching algorithm that provides delay guarantees while being efficient and practical to implement. The architecture in Fig. 2 suggests the following natural decomposition of the switching algorithm:

- demultiplexing: at every input i, deciding where to send each incoming cell.
- switching: for each of the k parallel switches, deciding on a matching, i.e., which cells to forward across the switch.
- multiplexing: at every output j, deciding which switch to read a cell from.

Before discussing the operation of this architecture, we describe why some simple approaches do not work.

A. Segmentation

The simplest approach one may consider is to segment each incoming cell into k segments, forward the segments in parallel across the switches, and reassemble the segments at the output.

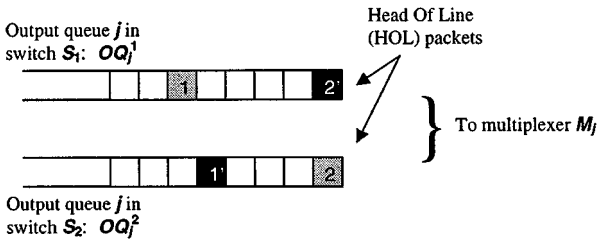

Fig. 3. Possibility of deadlock at the output.

Unlike what one might think, however, this approach does not eliminate the need for speedup. This is because each segment will now require $(1/k)$th the time of a complete cell, so a cell will have to be forwarded across the parallel switches in only $(1/k)$th of a cell slot. Thus, the time available for the switching algorithm also reduces by a factor of k, and k matchings will have to be computed per cell slot.

B. Rate Splitting

Yet another approach could be to split a flow among the parallel switches to divide its rate equally among them. If the parallel switches are allowed to forward cells independently, however, it is difficult to control the order in which cells of the same flow emerge at the outputs of the switches. This can lead to either deadlock or output overloading with FIFO output queues, as described below. For example, two cells that arrive at a given input and are sent to two different switches may experience different delays depending on the state of each switch and, thus, may arrive at the output in the wrong order. Even though it appears that this could be circumvented by controlling the order in which the output queues are read (that is, by determining at each cell slot the output queue containing the oldest cell of a flow and reading that cell), there could still be situations, such as the one depicted in Fig. 3, where no output queue can be read without violating the order of cells.

In Fig. 3, the cells at the head of output queue j in both parallel switches are the second cells of their respective flows. Thus, with FIFO output queues, it is not possible to deliver any cell at output j without violating the order of cells in a flow. Another solution could be to read the head-of-line (HOL) cells and temporarily store them to be delivered later. When the multiplexer has read deep enough into the output queues to be able to reconstruct the correct order of cells in a flow, the HOL cells stored earlier can be released in the correct order. Clearly, if this happens often, cell slots will be wasted without delivering cells at output j, causing the FIFO output queues to become overloaded and to grow indefinitely (see the Appendix). Of course, the above statement assumes that the output queues are FIFO and that a multiplexer cannot access more than one cell per cell slot. The choice of the FIFO restriction is based on the ease of implementation of FIFO queues. Restricting the multiplexer to at most one access per cell slot emanates from the need to have no speedup in any part of the parallel architecture. Both of these restrictions are reinforced by the fact the we do not allow for cell resequencing at the output.

Therefore, while on one hand our goal is to enable the switches to operate in a coordinated fashion, on the other it is

to avoid excessive bookkeeping of the type needed in [6] to emulate output queueing.

C. Basic Idea

The key idea is to first avoid the type of deadlock depicted in Fig. 3. Having achieved that, we focus later on how to provide the delay guarantees. We say that a cell C is older than a cell P if C arrives before P. In order to avoid the type of deadlock in Fig. 3, we consider the following two properties.

Definition 1 (Output Contention): In a single switch, two cells coming from different inputs and destined to the same output cannot be forwarded during the same cell slot (by the property of a matching, this is trivial when the switch has no speedup).

Definition 2 (Per-Flow Order): For any two cells C and P of the same flow, if C is older than P, then by the end of the cell slot during which P was forwarded, C would have been forwarded.

We will show that the two properties above are sufficient to ensure that, at an output j, the cells of any flow (i, j) can be read in order. We begin by defining this order more formally. In doing so, we define a partial order relation that we denote by $\prec_{\rm FIFO}$. The partial order relation $\prec_{\rm FIFO}$ is defined over all the cells that are residing at the output side of the switches. However, as it will be seen later from the definition of $\prec_{\rm FIFO}$, some cells might be left unordered by $\prec_{\rm FIFO}$. These are cells that are destined to different outputs or cells of different flows that are forwarded during the same cell slot. We will define the order relation $\prec_{\rm FIFO}$ in such a way that, if the *per-flow order* property is satisfied, it will induce the standard FIFO order on all cells pertaining to a single flow.

Definition 3 ($\prec_{\rm FIFO}$): For any two cells $C(i, j)$ and $P(k, j)$ at the output side, $C(i, j) \prec_{\rm FIFO} P(k, j)$ if

- the cell slot during which $C(i, j)$ was forwarded precedes the cell slot during which $P(k, j)$ was forwarded, or
- $i = k$, $C(i, j)$ is older than $P(k, j)$, and both were forwarded during the same cell slot.

Note that if $C(i, j) \prec_{\rm FIFO} P(i, j)$ and the *per-flow order* property is satisfied, then $C(i, j)$ is older than $P(i, j)$. More precisely, we have the following lemma.

Lemma 1: If the *output contention* and *per-flow order* properties are both satisfied, the following is true for every output j: At the end of a cell slot, either OQ_j^l is empty for all l or there exists a flow (i, j) such that its oldest cell $C(i, j)$ is at the head of OQ_j^l for some l.

Proof: If at the end of a cell slot, OQ_j^l is empty for all l, the lemma is true. So assume that, at the end of a cell slot, there is an l such that OQ_j^l is not empty. Since $\prec_{\rm FIFO}$ is an order relation, there must exist an l and an i such that OQ_j^l contains a cell $C(i, j)$ with the following property: there is no cell $P(k, j)$ at the output side satisfying $P(k, j) \prec_{\rm FIFO} C(i, j)$. We will prove that $C(i, j)$ is at the head of OQ_j^l and that $C(i, j)$ is the oldest cell of flow (i, j). We first prove that $C(i, j)$ is at the head of OQ_j^l. If a cell $P(k, j)$ is ahead of $C(i, j)$ in OQ_j^l, then by the *output contention* property, $P(k, j)$ was forwarded during a cell slot prior to the cell slot during which $C(i, j)$ was forwarded. By the definition of $\prec_{\rm FIFO}$, $P(k, j) \prec_{\rm FIFO} C(i, j)$, which is a contradiction. Next, we prove that $C(i, j)$ is the oldest cell of flow (i, j). If this is not so, note that by the *per-flow order* property, the oldest cell of flow (i, j), $P(i, j)$, must be at the output side, and in the worst case, must have been forwarded by the end of the cell slot during which $C(i, j)$ was forwarded. By the definition of $\prec_{\rm FIFO}$ and since $P(i, j)$ is older than $C(i, j)$, $P(i, j) \prec_{\rm FIFO} C(i, j)$, and we reach a contradiction again. ■

The above lemma implies that for every flow (i, j), whenever there are cells in the output queues for output j, a cell can be delivered at output j without violating the order of cells pertaining to flow (i, j). Therefore, this eliminates the deadlock situation described earlier and prevents the output queues from being overloaded. The *output contention* property is trivially satisfied when the switches have no speedup. Therefore, we will design our switching algorithm to satisfy the *per-flow order* property.

IV. Approach

To specify our approach, we will describe how we carry out the three steps outlined in Section III (demultiplexing, switching, and multiplexing). As motivated earlier, we will design our switching algorithm to satisfy the *per-flow order* property. We state the following definition which is needed in the rest of the paper.

Definition 4 (k-Parallel Switching): k-parallel switching is that where, during each cell slot, the switching algorithm computes only one matching M and applies it in all k parallel switches.

We start by describing the demultiplexer operation.

A. Demultiplexer Operation

To distribute the incoming cells among the k parallel switches, the demultiplexer follows a special demultiplexing strategy, which we call *minimum length* demultiplexing, as defined below:

Definition 5 (Minimum Length Demultiplexing): Demultiplexer D_i sends a cell destined for output j to a switch l with a minimum number of cells in VOQ_{ij}^l at the end of the cell slot preceding the current cell slot.

We now prove that this strategy together with k-*parallel* switching ensures that the k oldest cells for each flow (i, j) are always in distinct switches. We start with a simple lemma.

Lemma 2: If *minimum length* demultiplexing and k-*parallel* switching are used, then at the end of a cell slot, the lengths of VOQ_{ij}^l and VOQ_{ij}^s differ by at most 1 for any two switches l and s.

Proof: The proof is by induction on the number of cell slots.

Base Case: The lemma is trivially true at a fictitious cell slot before the beginning of the first cell slot.

Inductive Step: Assuming that the lemma is true at the end of cell slot T, we will prove that it holds at the end of cell slot $T+1$. We focus on any two VOQs, VOQ_{ij}^l, and VOQ_{ij}^s, and we consider two cases:

Case 1: At the end of cell slot T, both VOQs were nonempty. k-*parallel* switching during time slot $T+1$ will decrease the

length of both VOQs by the same amount (by either 0 or 1). If no cell is sent to either one of the VOQs during cell slot $T+1$, then the lemma holds at the end of cell slot $T+1$. Otherwise, a cell is sent to one of the VOQs say VOQ_{ij}^{l}. By the *minimum length* demultiplexing, we know that at the end of cell slot T, the length of VOQ_{ij}^{l} was at most that of VOQ_{ij}^{s}. Therefore, adding one cell to VOQ_{ij}^{l} will not violate the lemma.

Case 2: At the end of cell slot T, at least one VOQ, say VOQ_{ij}^{l}, was empty. Then we know by the lemma that VOQ_{ij}^{s} must contain at most one cell. If a cell $C(i, j)$ is sent during cell slot $T+1$ to either VOQ_{ij}^{l} or VOQ_{ij}^{s}, then by the *minimum length* demultiplexing it must be sent to VOQ_{ij}^{l}. Therefore, at the end of cell slot $T+1$, the length of both VOQs is at most 1, and the lemma holds. ∎

Using Lemma 2, we can now prove the following lemma:

Lemma 3: If *minimum length* demultiplexing and *k-parallel* switching are used, then for any flow, at the end of a cell slot, either all cells at the input side are in distinct switches or the k oldest cells at the input side are in distinct switches.

Proof: If at the end of a cell slot T, there is some VOQ_{ij} that is empty, then by Lemma 2, VOQ_{ij}^{l} has length at most 1 for all l, and, hence, all cells at the input side are in distinct switches. If at the end of a cell slot T, no VOQ_{ij} is empty, then for the k oldest cells at the input side not to be in distinct switches, it must be that some VOQ_{ij}, say VOQ_{ij}^{l}, contains two of the k oldest cells C_1 and C_2, and another VOQ_{ij}, say VOQ_{ij}^{s}, contains a cell C_3 that is not among the k oldest cells. Without loss of generality, C_3 is the head of VOQ_{ij}^{s} by the end of cell slot T. Let T_0 be the cell slot during which C_3 arrived to VOQ_{ij}^{s}.

Consider the end of cell slot $T_0 - 1$. Since only one cell $C(i, j)$, in this case C_3, can arrive during cell slot T_0, we know that at the end of cell slot T_0-1, both C_1 and C_2 were in VOQ_{ij}^{l}. Therefore, from the end of cell slot $T_0 - 1$ until the end of cell slot T, VOQ_{ij}^{l} was nonempty. Therefore, *k-parallel* switching implies that every time VOQ_{ij}^{s} was served by a matching, so was VOQ_{ij}^{l}. Since at the end of cell slot T, C_3 is at the head of VOQ_{ij}^{s}, all the cells that were in VOQ_{ij}^{s} at the end of cell slot $T_0 - 1$ must have been forwarded by the end of cell slot T. This means that at least that many cells, excluding C_1 and C_2, were also forwarded from VOQ_{ij}^{l}. Therefore, at the end of cell slot $T_0 - 1$, the lengths of VOQ_{ij}^{l} and VOQ_{ij}^{s} differed by at least two, which contradicts Lemma 2. ∎

Using Lemmas 2 and 3, we prove the main result of this section.

Theorem 1: If *minimum length* demultiplexing and *k-parallel* switching are used, then the *per-flow order* property is satisfied.

Proof: Consider a flow (i, j) and a cell slot T. If no cell $C(i, j)$ is forwarded during cell slot T, then the *per-flow order* property for flow (i, j) cannot be violated during cell slot T. Assume a cell $C(i, j)$ is forwarded during cell slot T. By Lemma 3, at the end of cell slot $T-1$, either all cells of flow (i, j) were in distinct switches or the k oldest cells of flow (i, j) were in distinct switches. Therefore, *k-parallel* switching cannot violate the *per-flow order* property during cell slot T. ∎

As a consequence, we can now prove that using *minimum length* demultiplexing and *k-parallel* switching cannot create the deadlock situation illustrated in Section III.

Corollary 1: If *minimum length* demultiplexing and *k-parallel* switching are used, then for every output j, at the end of a cell slot, either OQ_j^{l} is empty for all l or there exists a flow such that its oldest cell is at the head of OQ_j^{l} for some l.

Proof: Since the *output contention* property is trivially satisfied, the corollary is immediate from Lemma 1 and Theorem 1. ∎

The demultiplexers do not have to explicitly identify the VOQ with the minimum number of cells, as we can prove that each of the following strategies, when combined with *k-parallel* switching, is a *minimum length* demultiplexing.

Round Robin: In this strategy, each demultiplexer keeps N counters, one for each output. Each counter stores the identity of the switch to which a new cell for that output should be sent, and all counters start initially at 0. Every time the demultiplexer sends a cell for a particular output to the switch specified by the corresponding counter, it increments that counter modulo k. This has the nice property of dividing the rate of a flow equally among the k parallel switches. Moreover, as we will illustrate later in Section IV-D, this strategy will be useful for building a switch that supports a line speed that is k times the line speed of the individual switches.

Round-Robin Reset: This strategy is the *Round-Robin* strategy with a slight variation. For every flow (i, j), the system keeps track of the number of cells of that flow that are still residing at the input side of the switches. Whenever this number becomes 0, the counter at demultiplexer D_i that corresponds to output j is reset to 0. This strategy requires some extra information (to be kept either by the switching algorithm or by the demultiplexers) to correctly reset the counters of the demultiplexers. Moreover, it might require some synchronization between cell arrivals and the cell slots of the switch. As will be seen later, however, this strategy will actually allow the switching algorithm to keep less information for coordinating the operation of multiplexers at the output ports, and, in some cases, it also helps to reduce the amount of state information that the switching algorithm must consider for computing a matching.

Lemma 4: If *k-parallel* switching is used, then *Round-Robin* demultiplexing is a *minimum length* demultiplexing.

Proof: We will prove that for any flow (i, j), by the end of a cell slot T, either VOQ_{ij}s in all switches have the same length, or starting from a switch, we can find a round-robin order on the switches, S_1 to S_k, such that there exists $0 < l < k$, such that VOQ_{ij}^{l} is the last VOQ_{ij} that received a cell $C(i, j)$ by the end of cell slot T, the length of any VOQ_{ij}^{s} for $l < s \leq k$ is L, and the length of any VOQ_{ij}^{s} for $0 < s \leq l$ is $L + 1$. Note that proving the above claim proves the lemma since the next cell slot a cell $C(i, j)$ arrives, it will be sent to a VOQ_{ij} with the minimum number of cells, either because VOQ_{ij}s in all switches had the same length at the end of cell slot T, or because the cell is sent to $VOQ_{ij}^{(l+1)}$ by *Round-Robin* demultiplexing, which has a minimum number of cells. We prove the above claim by induction on the number of cell slots.

Base Case: The claim is trivially true at a fictitious cell slot before the beginning of the first cell slot since VOQ_{ij}s in all switches have the same length.

Inductive Step: The claim is true up to cell slot T. We will prove that it remains true for cell slot $T+1$. We are not going

to consider the interleaving in the operations of applying the matching and sending a cell to some VOQ, but one can show that this interleaving has no effect on the reasoning below.

If VOQ_{ij}s in all switches had the same length by the end of time slot T (or after the arrival of a cell during cell slot $T+1$), k-*parallel* switching implies that they will have the same length by the end of cell slot $T+1$. Moreover, by k-*parallel* switching, if a cell is forwarded from a VOQ^s_{ij} for $s > l$, a cell will be forwarded from a VOQ^s_{ij} for $s \leq l$. Therefore, the above claim will still be true after applying the matching.

If a cell $C(i, j)$ arrives during cell slot $T+1$ and VOQ_{ij}s in all switches had the same length by the end of cell slot T (or after applying the matching during cell slot $T+1$), then if $P(i, j)$ is sent to some VOQ^s_{ij} (which will have the maximum number of cells by the end of cell slot $T+1$), we set the order S_1 to S_k such that $S_1 = S_s$ and we make $l = 1$.

If a cell $C(i, j)$ arrives during cell slot $T+1$ and by the end of cell slot T (or after applying the matching during cell slot $T+1$), we had the order S_1 to S_k with some $l < k-1$, then we keep the same order and increment l by one. If $l = k-1$, then by the end of time slot $T+1$, VOQ_{ij}s in all switches will have the same length since $C(i, j)$ will be sent to S_k. ■

A similar proof can be constructed for *Round-Robin Reset* since *Round-Robin Reset* acts exactly like *Round Robin*, except that it resets the round-robin order for a flow whenever all cells of that flow have been forwarded. In the interval between two successive resets, therefore, *Round-Robin Reset* behaves exactly like *Round Robin* and, hence, satisfies *minimum length* demultiplexing.

B. Switching Operation

We showed how *minimum length* demultiplexing together with k-*parallel* switching can satisfy the *per-flow order* property, which (with the *output contention* property) ensures that, for every output j, it is possible to read a cell (if one is available) without violating the order of cells within a flow. In Section IV-C, we explain how, during each time slot, the multiplexer may identify the appropriate queue to read from. Our focus here is to consider how a matching M may be computed to achieve a bounded delay on every cell. We turn our attention first to a class of switching algorithms for the single switch setting that we call k-*serial* switching.

Definition 6 (k-Serial Switching): In a single switch setting, k-*serial* switching is one in which the switching algorithm applies a given matching M consecutively k times before computing and applying a new matching.

Our intention is then to show that any k-*serial* switching algorithm with a particular speedup can be emulated by a combination of *minimum length* demultiplexing and some k'-*parallel* switching algorithm, where we define emulation as follows.

Definition 7 (Emulation): If, using a k-*serial* switching algorithm, a cell C is forwarded across the single switch during a cell slot T, then using *minimum length* demultiplexing and some k'-*parallel* switching algorithm, the same cell would also have been forwarded across one of the k' parallel switches by the end of cell slot T.

In what follows, we will assume that cells arrive only at the beginning of a cell slot. This requirement can be realized by delaying an incoming cell until the beginning of the next cell slot, which increases the cell delay by at most one cell slot.

We first state the following simple lemma.

Lemma 5: For any real number S, if *minimum length* demultiplexing and $\lceil S \rceil$-*parallel* switching are used, and cell arrivals occur only at the beginning of a cell slot, then if M is the matching computed in cell slot T and $(i, j) \in M$, then either all the cells of flow (i, j) or the $\lceil S \rceil$ oldest cells of flow (i, j) are forwarded by the end of cell slot T.

Proof: If at the end of cell slot $T-1$, at least $\lceil S \rceil$ cells of flow (i, j) are at the input side, then the result is true by Lemma 3 applied at the end of cell slot $T-1$. If at the end of cell slot $T-1$, less than $\lceil S \rceil$ cells of flow (i, j) are at the input side, then assume, without loss of generality, that a cell $C(i, j)$ arrives at the beginning of cell slot T. By Lemma 3 applied at the end of cell slot $T-1$, and by *minimum length* demultiplexing, $C(i, j)$ will be sent to an empty VOQ_{ij}. Thus, at the beginning of cell slot T, there are at most $\lceil S \rceil$ cells of flow (i, j) at the input side, each being in a separate VOQ by Lemma 3. Therefore, $\lceil S \rceil$-*parallel* switching implies that all cells of flow (i, j) will be forwarded by the end of cell slot T. ■

Using Lemma 5, we can prove the following theorem.

Theorem 2: If cell arrivals occur only at the beginning of a cell slot, then any k-*serial* switching algorithm under a fractional speedup $S = k/c$ can be emulated using *minimum length* demultiplexing and an $\lceil S \rceil$-*parallel* switching algorithm.

Proof: Every c cell slots, the k-*serial* switching algorithm has exactly k matching phases, during all of which a matching M is kept constant. The $\lceil S \rceil$-*parallel* switching algorithm will run the k-*serial* algorithm in the background, and in doing so, it will compute the same matching M every c cell slots. We will prove the theorem by induction on the number of cell slots.

Base Case: The theorem is trivially true at a fictitious cell slot before the beginning of the first cell slot.

Inductive Step: By the end of cell slot T, all cells that were forwarded by the k-*serial* algorithm were also forwarded by the $\lceil S \rceil$-*parallel* algorithm. Consider cell slot $T+1$. Since the k-*serial* switching algorithm can have at most $\lceil S \rceil$ matching phases in every cell slot (speedup of S), this implies that if $(i, j) \in M$, then the number of cells of flow (i, j) that are going to be forwarded during cell slot $T+1$ by the k-*serial* algorithm cannot be more than $\lceil S \rceil$. By Lemma 5, if $(i, j) \in M$, then either all the cells of flow (i, j) or the $\lceil S \rceil$ oldest cells of flow (i, j) are forwarded during cell slot $T+1$ by the $\lceil S \rceil$-*parallel* algorithm. Therefore, if a cell $C(i, j)$ is forwarded during cell slot $T+1$ by the k-*serial* algorithm, and had not been forwarded by the $\lceil S \rceil$-*parallel* algorithm by the end of cell slot T, then it must be among the cells that will be forwarded during cell slot $T+1$. ■

Note that if S is an integer (which we can always assume to be true), then the $\lceil S \rceil$-*parallel* switching algorithm is a k-*parallel* switching algorithm because $S = \lceil S \rceil = k$. In this case, as a practical consideration, the k-*parallel* switching algorithm does not need to run the k-*serial* switching algorithm in the background. Instead, it reconstructs the state of the single switch from the k parallel switches. This is possible since in every cell slot both switching algorithms apply the same matching M an

equal number of times k (one in parallel and the other sequentially); therefore, by the end of a cell slot, the same cells which remain at the input side in the single switch also remain at the input side in the k parallel switches. This reconstruction of the exact state of the single switch requires also that the same cells are being read from the output queues in every cell slot by both algorithms (FIFO order). We know from Lemma 1 that this is possible (since k-*parallel* switching means that cells that are forwarded to the same output during a single cell slot pertain to the same flow, and, hence, all cells at an output are ordered by the $\prec_{\text{FIFO}}$ relation). Reconstructing the single switch from the k parallel switches, however, implies that the switching algorithm has to look at a large amount of state. At the end of this section, we will suggest a way to reduce the amount of state information that the k-*parallel* switching algorithm has to look at; namely, we will consider looking only at the state of the first switch.

Note also that since the $\lceil S \rceil$-*parallel* switching can exactly mimic the k-*serial* ($S = k$) algorithm when S is an integer, it can provide the same guarantees as the k-*serial* switching algorithm.

Below, we state some loose delay bounds that the emulation guarantees for every cell under a constant burst traffic [3]. In a constant burst traffic, the number of cells arriving at a given input port or destined to a given output port during an interval of time T is bounded by $\alpha T + B$ where B is a constant independent of time and $\alpha \leq 1$ is the loading of the switch. We define the arbitration delay of a cell as the time the cell remains in its VOQ. The following theorem states that if each output emulates a global FIFO queue, emulating a k-*serial* switching algorithm that guarantees a cell arbitration delay will also result in guaranteeing a total cell delay.

Theorem 3: If a k-*serial* switching algorithm under a constant burst traffic and a fractional speedup $S = k/c$ guarantees a cell arbitration delay D_A, then emulating that switching algorithm using *minimum length* demultiplexing and an $\lceil S \rceil$-*parallel* switching algorithm achieves a bounded delay of $(\lceil S \rceil + 1)D_A + B$ on every cell, where B is the traffic burst constant, provided that every output reads the cells in the $\prec_{\text{FIFO}}$ order.

Proof: By Theorem 2, we know that the $\lceil S \rceil$-*parallel* switching algorithm will guarantee a cell arbitration delay D_A. Therefore, at the end of a cell slot, the number of cells destined to an output j that are still at the input side cannot exceed D_A in any of the $\lceil S \rceil$ switches. Otherwise, at least one cell will be delayed for more than D_A cell slots at its input, implying that its arbitration delay will be greater than D_A. Consequently, at the end of a cell slot, the number of cells destined to output j that are still at the input side in all $\lceil S \rceil$ switches is at most $\lceil S \rceil D_A$. By Corollary 1, if there are cells waiting in some output queue OQ_j^l, then it is possible to deliver a cell at output j without violating the cell order of any flow. Therefore, as long as some OQ_j^l is not empty, output j delivers a cell. Consider a cell slot T in which some OQ_j^l becomes nonempty. At the end of cell slot $T-1$, the number of cells destined to output j that reside at the input side is at most $\lceil S \rceil D_A$, as argued above. If during c cell slots starting from cell slot T, some OQ_j^s is nonempty, then by the end of the c cell slots, output j will have delivered c cells. However, during the c cell slots, the total number of cells that could have been forwarded to some output queue of port j is at most $\lceil S \rceil D_A + c + B$; since at most $c + B$ cells destined to output j could have arrived during the c cell slots, by the property of the constant burst traffic. This means that the total number of cells that remain in the output queues of port j after the c cell slots is at most $\lceil S \rceil D_A + B$. This is true for any c; therefore, at the end of a cell slot, the number of cells in all output queues of port j is at most $\lceil S \rceil D_A + B$. As a result, since the output emulates a FIFO queue (with an $\lceil S \rceil$-*parallel* switching algorithm, all cells at a particular output are ordered by $\prec_{\text{FIFO}}$), once a cell arrives at the output side, it will be delivered within at most $\lceil S \rceil D_A + B$ cell slots, hence, achieving a bounded delay of $(\lceil S \rceil + 1)D_A + B$ on every cell. ■

If *Round-Robin* demultiplexing is used, then to achieve a bounded delay on every cell, we need not restrict the output to read cells in a global FIFO order. We only require that the output read cells of the same flow in order, which is a requirement we have imposed throughout the paper.

Theorem 4: If a k-*serial* switching algorithm under a constant burst traffic and a fractional speedup $S = k/c$ guarantees a cell arbitration delay D_A, then emulating that switching algorithm using *Round-Robin* demultiplexing and an $\lceil S \rceil$-*parallel* switching algorithm achieves a bounded delay of $(\lceil S \rceil + 1)D_A + B + (\lceil S \rceil - 1)(N-1)$ on every cell, where B is the traffic burst constant and N is the size of the switch, provided every output reads cells of the same flow in order.

Proof: The proof is similar to the proof of Theorem 3. We use the fact that at the end of a cell slot, the number of cells in all output queues of port j is at most $\lceil S \rceil D_A + B$. Assume a cell $C(i, j)$ remains in OQ_j^l for at least $\lceil S \rceil D_A + B + (\lceil S \rceil - 1)(N-1) + 1$ cell slots. By the end of the cell slot during which $C(i, j)$ was forwarded, OQ_j^l contained at most $\lceil S \rceil D_A + B$ cells including $C(i, j)$. Therefore, at least $(\lceil S \rceil - 1)(N-1) + 1$ cells, that were forwarded after $C(i, j)$, were delivered at output j before $C(i, j)$. These cells cannot pertain to flow (i, j) since cells of a flow are delivered in order. Therefore, at most N–1 flows can contribute to these cells. As a consequence, there exists a flow for which at least $\lceil S \rceil$ cells were forwarded after $C(i, j)$ and delivered at output j before $C(i, j)$. Since *Round-Robin* demultiplexing is used and the output reads cells of the same flow in order, it must be that one of these cells, say, P, was in OQ_j^l. But OQ_j^l is a FIFO queue and P was forwarded to it after $C(i, j)$. Therefore, P could not have been delivered at output j while $C(i, j)$ remains in OQ_j^l. ■

It remains for us to show the existence of k-*serial* switching algorithms that guarantee a cell arbitration delay under some speedup $S = k/c$. We will modify some existing switching algorithms that guarantee cell arbitration delay under some speedup S to make them k-*serial* switching algorithms, for any integer k.

1) Some k-Serial Switching Algorithms: In order to obtain k-*serial* switching algorithms, we convert existing switching algorithms for a single switch into k-*serial* switching algorithms, by simply modifying the existing algorithms to hold the matching that they compute constant for k times.[1] Our motiva-

[1]A similar idea was suggested in which a random matching is computed in every matching phase, but the matching is used only if it is better (in some sense) than the last used matching. Therefore, a matching might be held for a while before applying another matching.

tion is that the state of the switch cannot change substantially within a constant time k/S. Thus, holding the same matching for k times should possibly still be able to guarantee a cell arbitration delay.

We were able to prove this fact for several existing switching algorithms, such as the Oldest Cell First (OCF) algorithm [3], the Central Queue algorithm [8], and the Delayed Maximal Matching algorithm, an algorithm that we describe here in order to illustrate the point further.

Oldest Cell First: This algorithm is due to Charny *et al.* [3] and is a priority switching algorithm. The priority scheme used by this algorithm assigns higher priority to the VOQs holding older cells. Therefore, in every matching phase, the oldest cell that can still be forwarded is chosen. This is repeated until a maximal matching is obtained. This algorithm guarantees a bounded delay on every cell with $S = 2$ when $\alpha < 1$ under a constant burst traffic (see [3]), where α is the loading of the switch. The priority scheme of this algorithm guarantees that if a cell $C(i, j)$ is not forwarded, then either an older cell $P(i, k)$ is forwarded, or an older cell $P(k, j)$ is forwarded. Holding a matching M for k times starts to violate the above property for VOQ_{ij} only when some VOQ_{ik} (or some VOQ_{kj}) becomes empty while being served by the matching M. The above property will be violated at most $k-1$ times by a VOQ_{ik} (or a VOQ_{kj}) while VOQ_{ij} remains nonempty, since once VOQ_{ik} (or VOQ_{kj}) becomes empty, its incoming cells will definitely be more recent that that of VOQ_{ij} and this will be reflected by the priority scheme when the matching is recomputed after k cell slots. Therefore, holding the matching constant for k times will violate the above property for VOQ_{ij} at most $2(k-1)(N-1)$ times while VOQ_{ij} is nonempty (there are $2(N-1)$ VOQs that share either an input or an output with VOQ_{ij}). This is enough for the algorithm to still provide a delay guarantee (see [3]). The additional delay to the original delay will be $(2(k-1)(N-1))/(S-2\alpha)$.

Central Queue: This algorithm is due to Kam *et al.* [8]. The algorithm works by assigning credits to each VOQ_{ij} based of the rate of flow (i, j). A cell is admitted if it has credit. The credit is decremented by 1 whenever a cell is forwarded. The credit of a nonempty VOQ_{ij} represent the weight of (i, j). In every matching phase, the algorithm computes a 1/2-approximation of the maximum weighted matching, by repeatedly picking (i, j) with the largest weight until a maximal matching is obtained. This was proved to guarantee a bounded length of every VOQ under no speedup when the credit rate at each input and output is less then 1/2. As argued in [8], when a flow (i, j) is constantly backlogged, a bounded VOQ length L implies a bounded cell arbitration delay L/g_{ij}, where g_{ij} is the credit rate of flow (i, j). Using the same techniques in [8], one can prove that, with a speedup of 2 and a credit rate less than one, this algorithm also guarantees bounded queues.

During a constant time, the change in the credit assigned to a VOQ is bounded. Therefore, the change in the total weight of the maximum weighted matching is also bounded. Our matching, being a 1/2 approximation of the maximum weighted matching when first computed, when held for k times, cannot differ from the half weight of the maximum weighted matching by more than a certain bound. A problem arises, however, if a VOQ_{ij} with a large credit suddenly becomes empty. In that case, (i, j) is still considered part of the matching, while the matching is being held constant, and is contributing a large weight to the matching. However, that weight should not be counted in the matching because flow (i, j) is idle and no cells of flow (i, j) are being forwarded. Therefore, the weight of the real maximum weighted matching at that time might differ from the weight of the maximum weighted matching when our matching was computed, by as much as the credit of VOQ_{ij}. If flow (i, j) is constantly backlogged, however, when it becomes idle, the credit of VOQ_{ij} can be bounded. As a consequence, when all flows are constantly backlogged, the difference between the weight of the matching and the half weight of the maximum weighted matching is bounded at all times [the bound is $O(kN)$], and this is all what we need to keep the proof working (see [8]). Therefore, the VOQ length will still be bounded and a delay guarantee will be achieved when all flows are constantly backlogged. Note that in order to satisfy the requirement of Theorem 2, namely, that a cell arrival occurs only at the beginning of a cell slot, delaying an incoming cell until the next cell slot does not violate the condition that a flow is constantly backlogged.

Delayed Maximal Matching: This is a simple algorithm that we present to illustrate further the idea of holding the matching for a constant number of times. The switching algorithm waits for a time T until enough cells have accumulated in the VOQs. Then it forwards those cells in an interval of time T using successive arbitrary maximal matchings. During that interval of time, another set of cells would have accumulated, and the algorithm repeats. Therefore, the arbitration delay is T. One way of achieving this with a constant burst traffic is the following. The switching algorithm builds an $N \times N$ matrix A where a_{ij} represents the number of cells that arrived from input i to output j. The algorithm waits a time $T \geq B/(1-\alpha)$ where α is the loading of the switch and B is the traffic burst constant. Since the number of cells that arrive from an input or to an output during an interval of time T is at most $\alpha T + B$ (property of the constant burst traffic), the sum of entries of any row and any colon in the matrix A will be at most T. In that case, it can be shown that the switching algorithm can forward those cells in at most $2T$ maximal matchings. Therefore, with a speedup of $S = 2$, this is done in at most T cell slots. By that time, another matrix would have been computed and the same process is repeated again.

If we hold the matching for k times, every VOQ_{ij} will be served at most $k-1$ times while its $a_{ij} = 0$. We can show that this implies that the algorithm will need an extra $2(k-1)(N-1)$ matchings (or, equivalently, $(k-1)(N-1)$ cell slots with a speedup $S = 2$) to forward the cells during the interval of time T. For the process to work as before, we require that $\alpha T + B \leq T-(k-1)(N-1)$ or $T \geq (B+(k-1)(N-1))/(1-\alpha)$, which adds an extra delay of $((k-1)(N-1))/(1-\alpha)$.

2) Birkhoff–von Neumann Decomposition: A k-Parallel Switching Algorithm: Chang *et al.* [1] (see also [2]) have proposed an algorithm that is capable of providing delay guarantees for input-queued switches with no speedup. The algorithm consists of taking a static rate matrix and computing only once a static schedule in time $O(N^{4.5})$, based on a decom-

position result of Birkhoff and von Neumann. The schedule is a static list of matchings, corresponding to permutation matrices obtained from the decomposition of the rate matrix, and applied according to certain weights. In our context, we may utilize this algorithm in conjunction with *Round-Robin* demultiplexing which ensures identical rate matrices for all switches. Using this algorithm, k static schedules can be obtained based on the individual rate matrices. These static schedules will be identical since all switches will have the same rate matrix. Thus, as a natural consequence of this approach, the same matching will be applied in every cell slot in all of the parallel switches. For each individual switch, this provides comparable arbitration delay guarantees as the original algorithm of Chang, with the added advantage, that we can sustain a line speed that is now k times the speed at which the parallel switches operate. Note, however, that since each switch is now running at a slower speed, it is not possible to transmit cells at the line speed between the inputs and the switches and between the switches and the outputs. However, the technique described in of buffering cells at the demultiplexers and multiplexers can be utilized, causing only a small additive delay. This will be discussed with more detail in Section IV-D.

3) Reducing State Information: It can be shown that when the speedup $S = k$ is an integer, the k-*parallel* switching algorithm can reconstruct, from the state of all the k parallel switches, the state of the single switch running the k-*serial* switching algorithm. This requires, however, that the scheduler examine the state of each of the k parallel switches, and maintain a global state. It turns out that this global state requirement can actually be relaxed. For the single switch switching algorithms discussed above, only two kinds of state information are used: the oldest cell of each VOQ for OCF, and the length of each VOQ for Central Queue[2] and *Delayed Maximal Matching*.

By using *Round-Robin Reset* demultiplexing, the amount of state information needed can be greatly reduced for the OCF. For instance, it ensures that the oldest cell of every flow is always in the first switch. Thus, when using OCF and *Round-Robin Reset,* the algorithm needs only look at the state of the first switch to compute a matching.

For the Central Queue algorithm, the use of any *minimum length* demultiplexing ensures that, for every flow (i, j), the number L of all the cells at the input side is related to the number of cells L_1 in VOQ_{ij}^1 in the following way:

$$kL_1 - k < L < kL_1 + k.$$

Thus, if kL_1 is used as an approximation to L, the computation of the 1/2-approximation of the maximum weighted matching will be affected by at most a certain bound, which, as argued previously, will not hurt the delay guarantees for the Central Queue algorithm. Note that we are now using lengths of VOQs as the weights and not the credits (see footnote 2).

For the *Delayed Maximal Matching* algorithm, defining similarly a_{ij1} and using the upper bound $ka_{ij1} + k - 1$ as an approximation for a_{ij}, will result in serving a VOQ at most an additional bounded number of times while it is empty; a phenomenon that can be accommodated for in the same way described earlier for the k-*serial* version of the *Delayed Maximal Matching* algorithm, i.e., by increasing the delay after which the algorithm obtains a new matrix. The upper bound $ka_{ij1} + k - 1$ is used here because the algorithm needs to make sure that it is emptying all the matrix as described earlier.

Observe that state reduction is not an issue for the Birkhoff–von Neumann decomposition algorithm, because it only stores a precomputed schedule and so does not require any state information from the switches for its operation.

C. Multiplexer Operation

We have already shown that when using *minimum length* demultiplexing and k-*parallel* switching, it is possible for the multiplexer at an output port to always deliver a cell from the output queues of the k parallel switches in a way not to violate the order of cells pertaining to the same flow. The only question that remains is how a multiplexer M_j determines which output queue OQ_j^l to read the next cell from. This can be done in different ways. One way is to use a standard resequencing technique. Cells are tagged upon arrival to the switch with their arrival times. At the output side, the multiplexer incrementally sorts the tags of the HOL cells and chooses to read the one with the smallest tag. This requires additional access to the output queues which we assume not possible given that no speedup is available, especially since the tag value itself can grow as large as the total delay of a cell.

An alternative is for the switching algorithm to store this information and sort the HOL cells of all the queues. This, however, requires the communication of tags between the demultiplexers and the switching algorithm every time cells arrive. In addition, to avoid the use of unbounded tags, both of these approaches must address the issue of tag reuse.

We would like to avoid the use of the above resequencing techniques. A more efficient approach that uses *Round Robin* or *Round-Robin Reset* demultiplexing is the following. For each output j, the switching algorithm maintains a FIFO list L_j of tuples of the form (p, s) pertaining to successive cell slots during which a cell was forwarded to output j. Hence, for every such cell slot, p is the number of cells switched to output j during that cell slot, and s is the index of the switch that forwarded the oldest cell to output j during that cell slot (note that all cells switched to output j during that cell slot pertain to the same flow).

Therefore, during each cell slot for which some (i, j) belongs to the matching, the algorithm adds a (p, s) to L_j. The algorithm may easily obtain the information to do so from the demultiplexers. Each demultiplexer D_i stores the number of cells for a particular output that have arrived up to the current cell slot and are still remaining at the input side.

Upon applying a matching M, the switching algorithm communicates to demultiplexer D_i the index j of the output for which $(i, j) \in M$. The demultiplexer responds with the number of cells that will be forwarded to output j as a result of applying M (this is easy to determine since it is either all the cells or k cells by Lemma 5), and the index of the switch that contains

[2]Here, we say that the length of a VOQ instead of its credit because when a constant burst traffic where each flow has a rate and is constantly backlogged, the length of a VOQ differs from its credit by at most a constant. Alternatively, the Central Queue algorithm can use the credit of a VOQ and no other state information will be needed. But then, explicit knowledge of the rates is required.

the oldest such cell (also easy to determine with any of the two round-robin demultiplexing strategies described earlier). The total communication required between the demultiplexers and the switching algorithm is therefore $O(N \log N + 2N \log k)$.

Following this, demultiplexer D_i updates for every output j the number of cells of flow (i, j) remaining on the input side as well as the index of the switch that now contains the oldest cell of flow (i, j).

At the output side, each multiplexer M_j periodically retrieves from the switching algorithm a tuple (p, s) from which it learns the number of cells that must be read and the identity s of the switch from whose output queue the multiplexer must start reading the first cell of this round, and continues in a round-robin fashion (as a consequence of the round-robin demultiplexing). Therefore, the communication between the switching algorithm and the multiplexers is $O(2N \log k)$. Hence, the total communication with the switching algorithm is $O(N \log kN)$, which is within a constant factor of the $\Omega(N \log N)$ amount of communication needed for the switching algorithm to specify a matching in a single switch.

If we use *Round-Robin Reset* demultiplexing, then we know that the oldest cell of a flow is always in the first switch and, therefore, s is not needed.

Instead of requiring additional memory for the switching algorithm, we can use the memory of the switch itself, i.e., the output queues, in order to store the required information. This works for the case of *Round-Robin Reset* demultiplexing in the following way: Since the oldest cell of a flow is in the first switch, we only need to tag a cell $C(i, j)$ that is forwarded across the first switch with the number p of cells of flow (i, j) that are going to be forwarded during the current cell slot. At the output, the multiplexer retrieves this number when reading the cell in the first switch, and, hence, it knows how many cells to read before coming back to the first switch. This is efficient in terms of space since the tag length is $O(\log k)$ and only cells in the first switch need to be tagged. A difficulty with this approach is that we must tag cells upon forwarding them, which might not be straightforward to realize.

D. Supporting Higher Line Speeds

We now briefly describe how we can use parallel switches that run at a speed slower than the line speed. For this purpose, we assume that the line speed is some integer multiple, m, of the speed of a single switch.

The first thing to note is that each cell slot of a switch is now m times the original cell slot of the traffic (since the switches are m times slower). Thus, we will refer to the cell slots of the traffic by external cell slots, reserving the term time slots to denote the internal cell slots of the switches.

The second thing to note is that now a demultiplexer will not be able to send cells to a single switch in successive external cell slots, since each link can be accessed only once every cell slot, i.e., once every m external cell slots. Similarly, a multiplexer can access output queue OQ_j^l for output j once every m external cell slots.

We will assume the use of the *Round-Robin* demultiplexing strategy. Assume also that the number of parallel switches is m. We will describe how we can use the links that are running at $1/m$ the original line speed.

We will use m FIFO buffers running at the line speed in each demultiplexer and multiplexer. Each of the m FIFO buffers corresponds to one of the m switches. When a cell needs to be sent by demultiplexer D_i from input i to a switch, it is stored in a buffer of D_i corresponding to that switch. The cell at the head of the buffer is sent to the switch when the link is available. When only one cell can arrive at an input during a single external cell slot, an analysis of this technique appears in and illustrates that a buffer size of N is enough for each buffer of the demultiplexer. Moreover, each cell will be delayed at most N cell slots (i.e., mN external cell slots) at the input. Therefore, we can consider a new arrival pattern of cells at the input, where at a given time slot, only cells that arrived N cell slots prior to the current cell slot are considered present. This produces the original arrival pattern of cells delayed by mN external cell slots.

Similarly, when a cell needs to be delivered at output j by multiplexer M_j, it is stored, when the link is available, in a buffer in M_j corresponding to the switch being used to deliver that cell. The cell remains in that buffer until it can be delivered. Therefore, the m buffers of the multiplexer act as a resequencing buffer. As before, the analysis described in [7] yields a buffer size of N for each of the m buffers of M_j. Moreover, each cell will be delayed at most N cell slots (i.e., mN external cell slots) at the output. Therefore, by waiting mN external cell slots at the output, the same techniques for delivering cells described in the previous section are still valid, hence, making resequencing a simple operation.

In general, for a constant burst traffic with burst constant B, the buffer size will be $N+\lceil B/m \rceil$ and the delay of mN external cell slots will be replaced by $mN + B$.

The above buffering technique solves the problem of slow links with an additive delay of $2(mN + B)$. We now illustrate that with these m slow switches, we can still somehow emulate an *m-serial* switching algorithm running at the original line speed.

We consider the new arrival pattern at the input, which is the exact original arrival pattern delayed by $mN + B$ external cell slots.

The idea is similar to what Lemma 5 and Theorem 2 achieve. As before, the *m-serial* switching algorithm holds a matching M for m external cell slots, which is equal to one cell slot of the *m-parallel* switching algorithm. First we note that *minimum length* demultiplexing operates in every external cell slot now as opposed to every cell slot. Therefore, the number of cells in a VOQ at the end of an external cell slot might not be accurately defined since a matching requires m external cell slots (one cell slot) to complete. Conceptually, however, we can think of the matching taking effect only during the last external cell slot of a cell slot. Hence, *minimum length* demultiplexing reflects the correct number of cells in the VOQs as viewed by the demultiplexers in each external cell slot. Since in our setting, *Round-Robin* demultiplexing is a *minimum length* demultiplexing as proved earlier, and since *Round-Robin* demultiplexing does not rely on the number of cells in the VOQs, regardless of how the matching is carried during a cell slot, we will still have the same results as before. Namely, Lemma 5 will

still be true, and, hence, if (i, j) is in the matching M, then either all cells of flow (i, j) or the m oldest cells of flow (i, j) will be forwarded by the end of a cell slot. With a proof similar to the one for Theorem 2, and since the *m-serial* algorithm can forward at most m cells every m external cell slots, we conclude that the *m-parallel* algorithm emulates the *m-serial* algorithm up to an additive constant $m-1$; the reason being that a cell that is forwarded with the *m-parallel* algorithm during a cell slot T might have been forwarded by the *m-serial* algorithm during any of the m external cell slots that correspond to cell slot T. Therefore, if the *m-serial* switching algorithm guarantees an arbitration delay D_A external cell slots, the *m-parallel* switching algorithm will guarantee an arbitration delay of $D_A+m-1+mN+B=D_A+m(N+1)+B-1$ external cell slots.

V. Conclusion

We suggested a scheme that eliminates the need for speedup by using $k=\lceil S\rceil$ parallel input–output-queued switches with no speedup, where S is the speedup of the original switch. The key to our approach was to apply the same matching in all the parallel switches. By adapting existing switching algorithms for the single switch setting to hold their matching constant for a number of times, we were able to apply the same matching in all switches, and guarantee a bounded delay on every cell. In addition, both demultiplexing and multiplexing at the inputs and outputs, respectively, could be done using $O(N\log kN)$ amount of communication between the switching algorithm and the parallel switches. This is to be compared to the $\Omega(N\log N)$ amount of communication needed in a single switch for the algorithm to specify a matching. We also suggested some heuristics that reduce the amount of state information that the switching algorithm needs to look at in order to compute a matching, resulting in the algorithm looking only at the state of the first switch. Our approach offers the advantage of using a constant number of parallel layers. This was not the case in [6] and [7], which emulate output queueing for a high line speed using $O(N)$ output-queued switches running at lower speed with no memory speedup. While this dependence on N can be eliminated by replacing the output-queued switches with input–output-queued switches [4], the algorithm for emulating an output-queued switch becomes more complicated and much less practical to implement. Moreover, our approach makes use of FIFO queues only, whereas the approach outlined in [4] requires the use of non-FIFO queues. The bandwidth requirement of the architecture proposed here is kNR where R is the line speed. The authors of [7] succeeded in reducing this bandwidth requirement to NR only at the expense of allowing cells to arrive in an out-of-order fashion with a bounded delay of $O(N^2)$.

Appendix

The following scenario illustrates why simple rate splitting might not achieve 100% throughput. Consider the case of two switches. The splitting can be done by deploying a round-robin policy that, for each flow, decides where to forward the next cell of that flow. Both switches will receive half the original rate for every flow. Each switch runs a switching algorithm that achieves 100% throughput with a speedup of 2. It can be shown that when fed with data at half the original rate, each switch will appear to operate at a speedup of 2 and should be able to achieve 100% throughput had the order of cells been of no concern. The example will show that if we wait to reorder cells, the number of transmitted cells falls consistently short of the number of arriving cells, leading to overloading of the output. We use two 14×14 switches. We also use the OCF algorithm which favors older cells in the switch to be forwarded. Our example consists of a repeating sequence of ten cell slots. In the Fig. 4, we show the cell arrivals for both switches during the ten cell slot time. A dot represents a cell arrival during the particular cell slot.

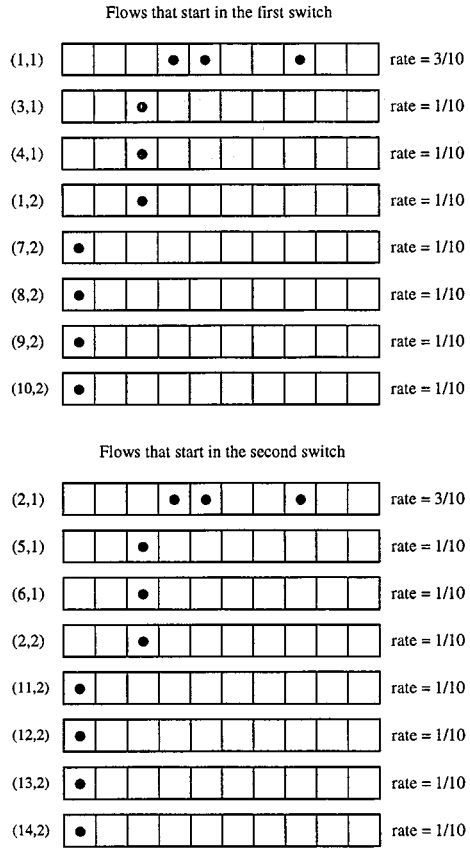

Fig. 4. Example of cell arrivals in a ten cell slot period.

We will refer to the first and second cells of flow (1, 1) by C_1 and C_2, respectively. Similarly, we will refer to the first and second cells of flow (2, 1) by P_1 and P_2, respectively. Note that C_2 and P_2 will be forwarded to switch 2 and switch 1, respectively, by the round-robin policy. Note also that the number of cells arriving for every flow is odd and, therefore, by the end of the tenth cell slot, each flow will start over in the other switch. Using OCF independently in each switch, we will force C_2 and P_2 to be forwarded before C_1 and P_1, thus, creating the deadlock situation described earlier.

In the first cell slot, only cells going to output 2 are received. Each switch will choose one of the cells going to output 2 to be forwarded. In the second cell slot, only cells going to output 2 are present in each switch because no arrivals occur; therefore, as before, each switch chooses to forward a cell from among the cells going to output 2.

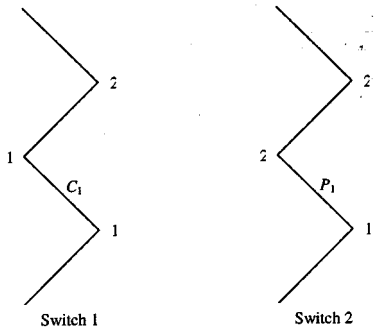

Fig. 5. Two switches at the beginning of the fourth cell slot.

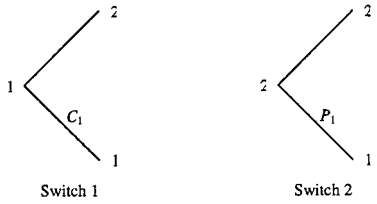

Fig. 6. Two switches at the end of the fourth cell slot.

In the third cell slot, each switch receives a new cell going to output 2, namely, $C(1, 2)$ in switch 1 and $C(2, 2)$ in switch 2, and some cells destined to output 1. Each switch chooses a cell going to output 1 and a cell going to output 2, where, by the OCF algorithm, these cells are different from $C(1, 2)$ and $C(2, 2)$, because $C(1, 2)$ and $C(2, 2)$ are the most recent cells.

In the fourth cell slot, switch 1 receives C_1 and switch 2 receives P_1. Therefore, at the beginning of the fourth cell slot, both switches have the configuration shown in Fig. 5, where an edge (i, j) represents a cell $C(i, j)$ in the switch.

By the selection of OCF, since $C(1, 2)$ and C_1 in the first switch are the most recent cells, they will not be forwarded during the fourth cell slot. A symmetric argument is valid for switch 2. Therefore, by the end of the fourth cell slot, we have the configuration of Fig. 6.

In the fifth cell slot, cells C_2 and P_2 arrive. This time they are sent to switch 2 and switch 1, respectively, because of the round-robin policy for each flow. At the beginning of the fifth cell slot, we have the configuration of Fig. 7.

Since $C(1, 2)$ is the oldest cell in switch 1, and $C(2, 2)$ is the oldest cell in switch 2, both of them will be forwarded. This means that C_1 in the first switch will not be forwarded. Similarly, P_1 in the second switch will not be forwarded. However, P_2 in the first switch will be forwarded because no other cells are blocking it. The same holds for C_2 in the second switch. By the end of the fifth cell slot, C_2 and P_2 are forwarded but not C_1 and P_1. By the end of the tenth cell slot, all cells are forwarded.

Since C_2 and P_2 were forwarded before C_1 and P_1, the only way to retrieve cells in order at the output is to read either C_2 or P_2, store it temporarily, and read it again when it can be delivered. Since we have ten cells to output 1 during ten cell slots, this implies that at least eleven reads are needed at that output in order to deliver all ten cells. However, the same pattern is being repeated every ten cell slots; therefore, output queue 1 will grow indefinitely. Note that although we are using the OCF algorithm for switching, the selection of cells in this example

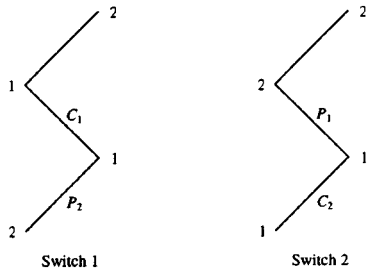

Fig. 7. Two switches at the beginning of the fifth cell slot.

satisfies the selection criteria for many other algorithms such as pure maximal matching, Central Queue [8], and lowest-occupancy-output-first algorithm (LOOFA) [9]. A simpler example that exhibits a similar scenario can be constructed; however, this particular example has the following advantageous properties.

- At most one cell arrives to a particular input during a cell slot.
- For every flow (i, j), the number of cells $A_{ij}(t)$ that arrive up to time t is bounded as follows:

$$rt - 1 \leq A_{ij}(t) \leq rt + 1$$

where r is the rate of the flow.

- The maximum aggregate rate at a port is 1; however, it can be reduced to be strictly less than 1 (namely, 20/21) by repeating the above pattern twice and inserting an idle slot after that. By doing so, we still have the inequality above. The output will still be overloaded since it has to make $22 = (11 + 11)$ output reads every $21 = (10 + 10 + 1)$ cell slots.

These properties imply that rate splitting combined with the algorithms mentioned above (which normally provide a cell delay bound with a rate less than 1/2, or a speedup of 2 and a rate less than 1) is not guaranteed to achieve 100% throughput even with the most restrictive input pattern, where the switches are operated independently. Note that the main problem here is not the size of the input queues but rather that of the FIFO output queues, and this is due to the possibility of misordering of cells of the same flow sent through different switch fabrics, which is not present in a single switch setting.

REFERENCES

[1] C.-S. Chang, W.-J. Chen, and H.-Y. Huang, "On service guarantees for input-buffered crossbar switches: A capacity decomposition approach by Birkhoff and von Neumann," in *Proc. 7th Int. Workshop Quality of Service (IWQoS 1999)*, London, U.K., pp. 79–86.

[2] C.-S. Chang, D.-S. Lee, and Y.-S. Jou, "Load balanced Birkhoff–von Neumann switches," in *Proc. IEEE Workshop High Performance Switching and Routing (HPSR 2001)*, Dallas, TX, pp. 276–280.

[3] A. Charny, P. Krishna, N. Patel, and R. Simcoe, "Algorithms for providing bandwidth and delay guarantees in input-buffered crossbars with speedup," in *Proc. 6th Int. Workshop Quality of Service (IWQoS 1998)*, Napa, CA, pp. 235–244.

[4] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with combined input–output-queued switches," *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1030–1039, June 1999.

[5] J. G. Dai and B. Prabhakar, "The throughput of data switches with and without speedup," in *Proc. IEEE INFOCOM 2000*, Tel Aviv, Israel, pp. 556–564.

[6] S. Iyer, A. Awadallah, and N. McKeown, "Analysis of a cell switch with memories running slower then the line rate," in *Proc. IEEE INFOCOM 2000*, vol. 2, Tel Aviv, Israel, pp. 529–537.

[7] S. Iyer and N. McKeown, "Making parallel packet switches practical," in *Proc. IEEE INFOCOM 2001*, Anchorage, AK, pp. 1680–1687.

[8] A. Kam, K.-Y. Siu, and R. Barry, "A cell-switching WDM broadcast LAN with bandwidth guarantee and fair access," *J. Lightwave Technol.*, vol. 16, pp. 2265–2280, Dec. 1998.
[9] P. Krishna, N. S. Patel, and A. Charny, "On the speedup requirement for work-conserving crossbar switches," *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1057–1069, June 1999.
[10] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," in *Proc. IEEE INFOCOM 1996*, vol. 1, San Francisco, CA, pp. 296–302.
[11] A. Mekkittikul and N. McKeown, "A starvation-free algorithm for achieving 100% throughput in an input-queued switch," in *Proc. Int. Conf. Computer Communication and Networking (ICCCN 1996)*, pp. 226–231.
[12] ——, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *Proc. IEEE INFOCOM 1998*, vol. 2, San Francisco, CA, pp. 792–799.
[13] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input-queued switches," in *Proc. IEEE INFOCOM 1998*, vol. 2, San Francisco, CA, pp. 533–539.

Saad Mneimneh was born in 1973 in Beirut, Lebanon. He received the B.E. degree in computer and communication engineering from the American University of Beirut (AUB) in 1995, the S.M. degree in information technology from the Massachusettes Institute of Technology (MIT), Cambridge, in 1997, and the Ph.D. degree in communication networks from MIT in 2002.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas. His research interests cover the algorithmic aspects of various networking problems including switching and routing, as well as approximation algorithms and online algorithms for problems related to deflection routing and unsplittable flows.

Vishal Sharma (S'92–M'98–SM'01) received the B.Tech degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1991, and the M.S. degrees in signals and systems and in computer engineering and the Ph.D. degree in electrical and computer engineering from the University of California at Santa Barbara (UCSB) in 1993 and 1997, respectively.

From 1997 to 1998, he was a Postdoctoral Researcher at the Multi-Disciplinary Optical Switching Technology (MOST) Center, UCSB. From 1998 to 2000, he was with the Tellabs Research Center, Cambridge, MA, where he worked on the analysis, architecture, and design of high-speed backbone routers, models for IP quality of service, and extensions to multiprotocol label switching (MPLS) for optical WDM and TDM networks. He is currently President and Principal Consultant at Metanoia, Inc., Mountain View, CA, specializing in technical consulting for telecom chip, system, and software vendors and service providers. He focuses on system and network analysis, detailed software and hardware system design and architectural tradeoffs, product development, technology and patent strategies, and knowledge enhancement for Metanoia's clients. He has eight patents in process in the areas of high-speed switch architectures, switch/router scheduling algorithms, IP protocols, MPLS and optical recovery, optical routing, and the dynamic control of SONET networks. He is on the Conference Advisory Board for MPLSCon and the Scientific Committee for the MPLS World Congress 2003, and is Co-Chair for the Industry Watch Program at Opticomm 2002. His current reserach interests include the design and analysis of high-speed communication networks, advanced IP routing and signaling protocols, dynamic optical networks, and models and techniques for service differentiation in IP networks.

Kai-Yeung Siu received the B.S. degree (*summa cum laude*) in mathematics and computer science from New York University, New York, NY, and the B.Eng. degree (*summa cum laude*) in electrical engineering from The Cooper Union, New York, NY, both in 1987. He received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1988 and 1991, respectively.

From 1989 to 1990, he was a Research Student Associate with the IBM Almaden Research Center, San Jose, CA. From 1991 to 1995, he was Assistant Professor of electrical and computer engineering at the University of California, Irvine. He joined the Massachusetts Institute of Technology (MIT), Cambridge, in 1996, where he is currently Associate Professor and recipient of the d'Arbeloff Career Development Chair. He is with the d'Arbeloff Laboratory for Information Systems and Technology of Mechanical Engineering and is also affiliated with the Laboratory for Information and Decision Systems of Electrical Engineering and Computer Science. He is the Research Director of the MIT Auto-ID Center, an industry-funded center which develops next-generation automatic identification systems with e-commerce applications. He has published over 100 research papers in the areas of optical networking, wireless communications, Internet routing and congestion control protocols, parallel and distributed algorithms, and computational complexity theory. He has served as a Consultant for major Internet equipment vendors and service providers.

Dr. Siu has served on the Editorial Board of the IEEE TRANSACTIONS ON NETWORKING. He received a National Science Foundation Young Investigator Award in 1993, the UC Irvine Distinguished Assistant Professor Award in 1995, the IEEE Browder J. Thompson Memorial Prize Paper Award in 1997, and the Best Paper Award of the SPIE Conference on All-Optical Networking in 1998.