

An $O(\log^2 N)$ parallel algorithm for output queuing

Amit Prakash Sadia Sharif Adnan Aziz
Department of Electrical and Computer Engineering
The University of Texas at Austin
{prakash, sharif, adnan}@ece.utexas.edu

Abstract—Output queued switches are appealing because they have better latency and throughput than input queued switches. However, they are difficult to build: a direct implementation of an $N \times N$ output-queued switch requires the switching fabric and the packet memories at the outputs to run at N times the line rate. Attempts have been made to implement output queuing with slow components, e.g., by having memories at both inputs and outputs running at twice the line rate. In these approaches, even though the packet memory speed is reduced, the scheduler time complexity is high — at least $\Omega(N)$. We show that idealized output queuing can be simulated in a shared memory architecture with $(3N - 2)$ packet memories running at the line rate, using a scheduling algorithm whose time complexity is $O(\log^2 N)$ on a parallel random access machine (PRAM). The number of processing elements and memory cells used by the PRAM are a small multiple of the size of the idealized switch.

I. INTRODUCTION

A packet switch needs to be able to buffer packets because of contentions for output links. This buffering can be at the input port, at the output port, in the switching fabric, or in shared memories. The first and second cases are referred to as *input queuing* and *output queuing*, respectively [1].

Output queued switches are appealing because they have better latency and throughput than input queued switches. However, a direct implementation of an output queued switch needs to run the switching fabric and the buffer memory at N times the line speed for an $N \times N$ switch (since at the start of a cycle, all packets at an input port may be destined to the same output port). Thus input queuing is preferred for implementation reasons, and considerable effort has been devoted to overcoming its limitations, e.g., the development of virtual output queuing to overcome head-of-line blocking [2].

It is natural to ask if it is possible to build a switch whose external behavior is identical to an idealized output-queued switch using slower components. Chuang *et al.* [3] define a switch S as simulating output queuing, if, given identical inputs, the departure time of every packet from S is the same as that from the output queued switch. They showed that a switch using queues which can support 2 reads and 2 writes per cycle at both the input and output ports can be scheduled to simulate output queuing. However, computing this schedule itself takes time complexity $\Omega(N)$, which renders it impractical for large switches. Iyer *et al.* [4] showed how to build fast routers by running slower ones in parallel; they use the switch of Chuang *et al.* as a component, and their scheduler also has a high complexity.

Our switch uses a shared memory architecture with a maximum of $(3N - 2)$ packet memories running at the line rate. It emulates FCFS output queuing using a switch scheduling algorithm with time complexity $O(\log^2 N)$ on a parallel random access machine (PRAM). This scheduling algorithm can be adapted for an output queued switch which provides per class quality of service.

The remainder of this paper is structured as follows: in Section II we describe our switch architecture and describe a schedule for this architecture which results in the switch simulating first-come first-served (FCFS) output queuing. In Section III we review some graph theory which forms the basis of a parallel algorithm for computing the schedule efficiently. This parallel algorithm is discussed in Section IV and has time complexity $O(\log^2 N)$ on a CREW PRAM. In Section V we discuss how the scheduler can be modified for a generalized output queued switch. In Section VI we discuss implementation issues. We conclude with a summary of our results and future work in Section VII.

II. FCFS OUTPUT QUEUED SWITCH

For simplicity, we will consider FCFS output queuing first. We assume packets arriving at the same cycle for the same output port are served according to a fixed pre-assigned order, e.g., their input port ids. Henceforth, we will use the term output queuing in this restricted sense. We will consider the case of general output queuing later in Section V.

The combined input and output queued switch discussed by Chuang *et al.* [3] emulates output queuing by using a speedup of 2. It is natural to ask whether the schedule they describe can be computed efficiently on a parallel computer. Unfortunately, the answer thus far is no — computing their schedule involves solving a stable marriage problem, and the best known parallel algorithm for the stable marriage problem, due to Feder, Megiddo, and Plotkin [6], has complexity $O(\sqrt{n} \cdot \log^3 n)$ and uses n^4 processors. Furthermore, this algorithm is extremely complicated: it is based on interior point methods for linear programming. Thus it is neither theoretically efficient (i.e., does not have $O(\text{polylog}(n))$ complexity, nor of practical significance.

A. Switch architecture

Figure 1 depicts the shared memory switch architecture that is the focus of this work. The set of input ports is connected via a crossbar to a set of M memories; these memories are connected to the set of output ports through another crossbar. Let

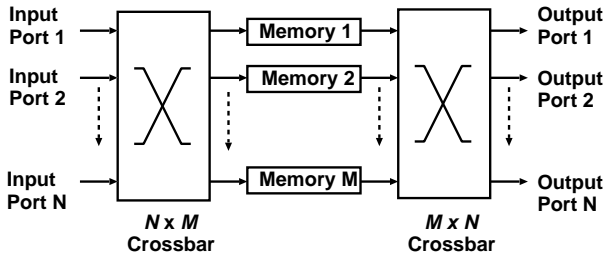


Fig. 1. Switch architecture.

the output queues from the switch we want to simulate be of size K . We will choose the M memories to be of size K each. We assume that $K \geq N$, i.e., the size of the output queues in the ideal switch is at least equal to the number of input and output ports.

Packets are assumed to arrive synchronously at the input ports, with a fixed delay between arrivals. We will refer to this delay as a *cycle*. In each cycle, the switch functions by moving packets from the input ports to the memories, and then from the memories to the output ports. When the switch first starts operation, all the memories are empty. For simplicity, we will assume that the bandwidth of each memory is equal to twice the line rate. Therefore each memory can sustain one read and one write operation per cycle.

B. Switch Scheduling Algorithm

Each arriving packet on an input port faces two kinds of *conflicts*. Packets arriving at the same time cannot be written to the same memory. We refer to these as *arrival conflicts*. Since there are N input ports, the maximum number of arrival conflicts a packet can have is $(N - 1)$. *Departure conflicts* occur if multiple packets in the same memory are accessed simultaneously by different outputs. Because there are N outputs the cardinality of this set of conflicts is also $(N - 1)$. Thus by the pigeon hole principle a scheduler needs only $(2N - 1)$ memories to avoid conflicts at both inputs and outputs. However, the parallel algorithm presented in Section IV may need additional memories in order to be able to compute the matching efficiently.

To emulate an ideal output queued switch, the scheduler has to ensure that both input and output conflicts can be resolved. Arrival conflicts can be avoided if incoming packets are written to distinct memories. Departure conflicts can be avoided if output ports read packets from different memories. Both conflicts are avoided as follows:

- 1) At the start of each cycle compute the departure time of each newly arrived packet.
- 2) For each incoming packet, determine which memories are compatible with the packet, i.e., do not currently store a packet with the same departure time.
- 3) Compute a matching of newly arrived packets to memories which pairs each packet with a distinct memory with which it is compatible.

Counters associated with each output queue keep a count of the number of cells in the queue. They are incremented each time a cell arriving for the output is placed in the memory. They are decremented each time a cell is transmitted by the output port.

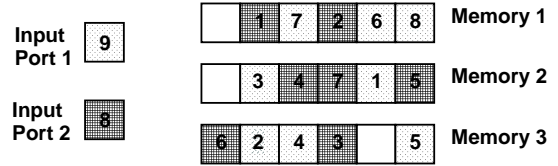


Fig. 2. Snapshot of the cells in the memories and input ports of a 2×2 switch.

We will find it convenient to have a counter $curr_time$, which is initialized to 0, and is incremented by 1 at the start of each new cycle. The $curr_time$ counter is added to the queue length of an output port to determine the departure time of each arriving packet. If multiple packets arrive for the same output, ties are broken according to input port ids and each packet is given a distinct departure time.

The scheduler switches packets across the first crossbar to the memories according to the computed schedule. Each memory maintains separate logical queues for each of the outputs. When an output queue becomes full, incoming packets for that output are discarded. Packets whose departure time is the current time, are switched from memories to output ports through the second crossbar.

Lemma 1: The switch in Figure 1 simulates output queuing under the schedule described above.

Proof: Since packets are read out at the time they would have departed from the corresponding output, it suffices to show that no packet is ever stalled at an input port or memory. Since at most one packet in a memory can have a specific departure time for a given output port, and only one packet with a specific departure time is ever written to a given memory, no packet is ever stalled at a memory.

To show no packet is ever stalled at an input port, we need to show that there always exists a matching, which pairs each newly arrived packet to a distinct compatible memory. First note that since there are N output ports, if a new packet arrives at a given input port with a departure time δ , there can be at most $N - 1$ packets already in the memories having departure time δ . Consequently, there are at least $M - (N - 1)$ memories which do not have a packet with departure time δ . If $M \geq 2N - 1$, then at least N memories do not have a packet with departure time δ .

We claim that a memory which does not have a packet whose departure time is δ cannot be full. The proof is by contradiction: suppose it was full. Then since it does not contain a packet whose departure time is δ , there are at most $K - 1$ possible departure times which are represented in the memory. However, since the memory is full, it contains K packets. By the pigeon hole principle, two of these packets share the same departure time, which violates the principle that packets are assigned only to compatible memories. Thus any newly arrived packet is compatible with at least N memories. Since there are N arriving packets, we can iteratively match packets to compatible memories to build the desired matching. ■

Consider a simple 2×2 switch with 3 memories. Figure 2 shows the contents of the memories and the packets arriving at the input ports at a particular time. The numbers represent the timestamps of the cells. Note that each cell in the same memory has a unique timestamp which ensures that only one output port

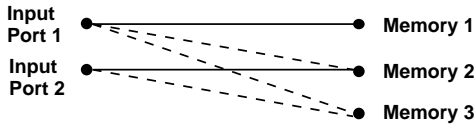


Fig. 3. G_c for the 2×2 switch in Figure 2.

will read from a memory in one cycle. The arriving cells have to be placed in the memories which do not have the cells with the same time stamp. At the same time each cell must be placed in a different memory since only one write to a memory is permitted in a cycle. The cell at input port 1 has a timestamp of 9 and can be placed in any of the memories. The cell at input port 2 has a timestamp of 8 and cannot be placed in memory 1. It can be placed in either of the other two memories. The scheduler finds a valid match and pairs input port 1 with memory 1 and input port 2 with memory 2.

III. SCHEDULING AND GRAPH THEORY

The problem of computing a matching of newly arrived packets to memories, which pairs each packet with a distinct compatible memory, can be formulated as a maximum matching problem on a bipartite graph [5, pages 600–604]. Specifically, the bipartite graph consists of N vertices corresponding to the input ports in one partition, and M vertices corresponding to the memories in the other partition. Keeping Figure 1 in mind, we will find it useful to refer to the first partition as being on the *left* and the second one as being on the *right*. We keep the edge (i, j) iff there is a newly arrived packet at input port i which is compatible with memory j . Call this graph G_c .

A. Matchings in bipartite graphs

A matching in G_c consists of a pairing of newly arrived packets to distinct compatible memories. By the counting argument in the proof of Lemma 1, we know that a matching exists in which every vertex on the left which has nonzero degree is matched. Hence by computing a maximum matching in G_c , we can obtain the desired assignment of packets to compatible memories. Figure 3 shows the graph G_c obtained for the 2×2 switch in Figure 2. The solid lines show the matching computed by the scheduler.

B. Edge coloring a bipartite graph

An *edge coloring* of an undirected graph G , is an assignment of colors to edges such that no two edges which are incident on the same vertex are assigned the same color. It is a nontrivial fact due to König [8, page 103] that a bipartite graph can always be edge-colored with exactly as many colors as the maximum degree of any vertex in the graph. Each of these colors forms a matching. Figure 4 shows a bipartite graph colored with two colors. The set consisting of edges of one color form a matching in the graph.

In general, there is no relationship between the minimum edge coloring of a bipartite graph and its maximum matchings. However, for our problem we can compute a maximum matching for G_c by computing a minimum edge coloring on a subgraph of G_c . Specifically, we can prune edges from all vertices

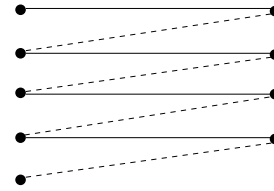


Fig. 4. Edge coloring in a bipartite graph.

on the left with nonzero degree so that each such vertex has degree exactly N ; call the resulting graph H_c . The vertices from the right partition in H_c subgraph still have degree at most N . Thus H_c can be edge-colored with exactly N colors. Now we claim that the collection of all edges which are colored a specific color, say κ , is a maximum matching. This follows from the fact that each left vertex which had nonzero degree now has degree N and so must be connected to an edge of color κ . The fact that no pair of edges in this collection share a vertex follows from the definition of edge-coloring.

IV. EFFICIENT PARALLEL IMPLEMENTATION

A. Parallel computing

First we note that every scheduling algorithm running on a uniprocessor will have $\Omega(N)$ time complexity, since at the very least, it will need to read the destination output port for packets at each of the input ports. Thus, the only way to perform scheduling in $o(N)$ time is to use parallelism.

The basic model of a parallel computer is the parallel random access machine (PRAM) [5]. This consists of a collection of shared memory cells and a set of processors. Each processor is a random access machine, i.e., a conventional uniprocessor. Computation takes place in sequential steps, wherein each processor reads from a shared memory cell, performs a local computation, and writes to a shared memory cell. Different processors can execute different instructions; each processor knows its own index. Reads, local operations, and writes take place at distinct phases within a cycle. There are variations on the amount of concurrent accesses that can be made to the shared memory cells. For our purposes, a Concurrent Read Exclusive Write (CREW) PRAM, where multiple processors can read from a cell, but only one can write to it, will suffice.

The standard criterion for classifying a parallel algorithm as being efficient is that its time complexity should be $O(\log^{k_1} n)$, where k_1 is some constant and n is the size of the problem instance, and the total number of processors and memory cells used should be $O(n^{k_2})$, where k_2 is some constant. The set of problems for which there is an efficient algorithm is denoted by NC , and is the parallel computing world's analog of P , the class of polynomial time solvable problems in the uniprocessor world.

A uniprocessor implementation of a scheduler computing a schedule as described in the previous section will take $\Omega(N)$ time. However, we will now show that the problem of computing the schedule given above is amenable to an efficient solution by a parallel computer. Specifically, we will show that the above schedule can be computed in $O(\log^2 N)$ time on a CREW PRAM, and that the number of processing elements and

memory cells used by this PRAM is a small multiple of the size of the output switch being simulated.

The PRAM will sit outside the switching fabric in Figure 1. We assume that at the start of each cycle, the destination output ports for arriving packets will be written into a pre-selected set of shared memory cells (a -1 is written if no packet arrives for that port). The PRAM will then compute an assignment from the N input ports to the M memories, and an assignment of memories to output ports and write these to a pre-selected set of shared memory cells. (The assignment will hold -1 if no packet is to be transferred.) These assignments will be used to program the crossbar to switch packets to memories. We now describe the implementation of the scheduler in detail.

B. Departure time computation

The departure time computation can be performed by a *prefix-sum* operation. Prefix computations are a standard operation on PRAMs. The computation takes an ordered set of n elements $\langle a_1, a_2, \dots, a_n \rangle$ and a binary associative operator \otimes , and returns the ordered set $\langle a_1, a_1 \otimes a_2, \dots, a_1 \otimes a_2 \otimes \dots \otimes a_n \rangle$. For prefix-sum computations, the binary associative operator is ordinary addition. For example, if the original sequence of numbers is $\langle 0, 1, 2, 3, 4 \rangle$, then the sequence of prefix sums is $\langle 0, 1, 3, 6, 10 \rangle$. If the ordered set is kept as an array or a linked list, the prefix-sum operation can be computed in $O(\log n)$ steps using n processors using a “pointer doubling” technique [5].

We reserve an N -wide array L of PRAM memory cells, which will be used to keep track of the departure times of the end-of-queue packet for each output port. We reserve an $N \times N$ matrix D of PRAM memory cells to help compute departure times of incoming packets. The entries in D are initialized to 0 at the start of each scheduling cycle. If input port i has a packet destined for output port k , it writes a 1 into $D[k][i]$. Updating L simply consists of adding $D[k][i]$ to $L[k]$ for each i . At the end of the cycle, each $L[k]$ which is nonzero is decremented by 1.

We then perform a parallel prefix-sum computation on each of the $D[k]$ arrays. Now $D[k][i]$ holds the number of 1s appearing in the array $D[k]$ up to and including the i -th entry, i.e., the number of packets destined to output port k . Since packets arriving in the same cycle for the same output port are served in order of their input port ids, the departure time for the packet at port i is $L[k] + D[k][i]$. If $L[k] + D[k][i] \geq \text{curr_time} + K$ the packet is dropped. We reserve an N -wide array δ of PRAM memory cells into which we write the departure times.

1) *Complexity analysis:* We need to do one instance of a prefix-sum computation for each output port in order to get departure times. Each of these prefix-sum computations is independent of the other, and so the departure time computation can be performed with N^2 processors in $O(\log N)$ time. Given N^2 processors, updating L , and clearing D are both constant time operations. In order to store the result of prefix computation we need N^2 memory locations. The total number of PRAM memory cells for the matrix D and the arrays L and δ is $N^2 + N + N = O(N^2)$.

C. Compatibility computation

We reserve a matrix C of $N \times M$ PRAM memory cells which will be used to represent the compatibility matrix, i.e., $C[i][j] = 1$ if the packet at input port i is compatible with memory j , and $C[i][j] = 0$ otherwise. We will later show that the number of memories M needed is no more than $(3N - 2)$.

We compute C as follows. We keep a $(3N - 2) \times K$ matrix of memory cells T , which is used to keep track of departure times represented in each memory. Specifically, $T[j][t \bmod K] = 0$ if memory j can accept a packet with departure time t , and $T[j][t \bmod K] = 1$ otherwise. (We never need to consider time slots greater than K in the future, since the maximum departure time of any cell in the output queued switch can be at most $K - 1$ more than the current time.) We have a distinct processor P_{ij} for each memory cell in the C matrix; processor P_{ij} writes a 1 in $C[i][j]$ iff $T[j][\delta[i] \bmod K] = 0$.

The matrix T is maintained by a set of $(3N - 2)$ processors $Q_1, Q_2, \dots, Q_{(3N-2)}$. In the next section, we will see how to compute a $3N - 2$ -wide vector B which encodes the matching of memories to input ports. At the end of the cycle, after this matching is computed, Q_j tests $B[j] \neq -1$, and if so, writes a 1 into $T[j][\delta[B[j]] \bmod K]$, indicating that the departure time of the packet at input port $B[j]$ is now not available in memory j . Processor Q_j then writes a 0 into location $T[j][\text{curr_time} \bmod K]$; this is to indicate that that location is now available.

1) *Complexity analysis:* The memory requirement is dominated by the C and T matrices, and is $O(N^2 + N \cdot K)$, which simplifies to $O(N \cdot K)$, since by assumption, $K \geq N$. The operations performed in computing C and in updating T are all constant time operations so the time complexity is $O(1)$. The number of processors is dominated by the $3N^2$ processors we use to compute C , and hence is $O(N^2)$.

D. Computing a matching from inputs to memories

By the counting argument in the proof of Lemma 1, we know that there exists a matching in which every vertex on the left which has nonzero degree is matched. Hence by computing a maximum matching in G_c , we can obtain the desired assignment of packets to compatible memories.

Unfortunately, the only known PRAM algorithms which have time complexity $O(\log^k N)$ for computing a maximum matching in bipartite graphs use randomization [7]; the existence of efficient deterministic parallel algorithms for maximum bipartite matching is an outstanding open problem in parallel computation. A randomized algorithm is not acceptable for our application, since we have to compute the schedule within a fixed amount of time, i.e., before the next set of packets arrive.

Luckily, we can use edge coloring of G_c to compute a matching deterministically in $O(\log^2 N)$ time. Let G be a bipartite graph on n vertices having the property that the maximum degree Δ of any vertex in G is a power of two. Lev, Pippenger, and Valiant [9] developed a PRAM based parallel algorithm for edge coloring such graphs which has time complexity $O(\log n \cdot \log \Delta)$, and uses $O(n^2)$ processors.

We cannot directly use their procedure on the graphs G_c or H_c , since N may not satisfy the requirement that the maximum degree of any vertex is a power of two. However, we

will show how to construct a subgraph G_c^* of G_c which has the property that all the vertices from the left partition of G_c which had nonzero degree will now have degree $\Gamma \geq N$, where Γ is a power of two, i.e., $\Gamma = 2^\gamma$ for some integer γ .

First we show that when $N \geq 2$, there exists an integer which is a power of two in the interval $[N, 2N - 1]$.

Proof: Consider the infinite sequence of integers $(2^1, 2^2, 2^3, \dots)$. Suppose for contradiction that there is no power of two in the interval $[N, 2N - 1]$. Let 2^l be the greatest integer in the sequence such that $2^l < N$. Then, by hypothesis we must have $2^{l+1} > 2N - 1$. However, since $2^l < N$, by multiplying both sides by 2, we have $2^{l+1} < 2N$. All together, $2N - 1 < 2^{l+1} < 2N$, which violates the integrality of 2^{l+1} . Hence there is a power of two in the interval $[N, 2N - 1]$. ■

By choosing M to be $(N - 1 + (2N - 1)) = (3N - 2)$, we are guaranteed that at least $(3N - 2) - (N - 1) = 2N - 1$ memories will be compatible with any given arriving packet. Thus, when $M = (3N - 2)$, there are at least $2N - 1$ edges incident to each nonzero degree vertex in the left partition in G_c , and at most N edges incident on each vertex in the right partition of G_c . Take Γ to be a power of two in the interval $[N, 2N - 1]$. We efficiently prune G_c to ensure that the all nonzero degree vertices have degree Γ as follows: we perform a prefix sum on the array $C[i]$ to compute a new array $C'[i]$. The entry $C'[i][j]$ is then set to 0 if $C'[i][j] > \Gamma$. Call the updated C matrix C^* ; then the graph corresponding to C^* is the desired graph G_c^* .

By König's theorem, we know that the edges of G_c^* can be colored using exactly Γ colors. Furthermore, since Γ is a power of two, this edge coloring can be performed using the procedure of Lev, Pippenger, and Valiant in $O(\log N \cdot \log \Gamma)$ time. Since $N \leq \Gamma \leq (2N - 1)$, the time complexity is $O(\log^2 N)$.

Since we are only interested in one color class, rather than a complete edge coloring of G_c^* , we can use an algorithm which, though based on Lev, Pippenger, and Valiant's edge coloring procedure, is significantly simpler to code. (However, its asymptotic complexity is identical to the algorithm of Lev, Pippenger, and Valiant.) Such an algorithm is described in Appendix A. By examining the complexity bounds in Sections IV-B.1 and IV-C.1 we obtain the following theorem:

Theorem 1: A CREW PRAM can be programmed to compute a schedule for the switch architecture in Figure 1 which simulates output queuing in time $O(\log^2 N)$; the PRAM uses $O(N^2)$ processors, and $O(N \cdot K)$ shared memory cells.

V. GENERALIZED OUTPUT QUEUED SWITCH

Our approach does not directly extend to output queued switches with multiple service classes and a general service discipline on the output ports, since we use the departure time stamps in an essential way. These can be computed when a packet arrives in a FCFS output queued switch, but the possibility of preemption precludes this with general output queuing. Before we embark on building a general OQ switch with QoS, we first answer the following question; can we build a FIFO switch without using timestamps for departure conflict resolution?

Instead of using the departure time of a packet, departure conflicts can also be avoided by ensuring that packets for an output are always *striped* across N memories. What this means

0	1	2	3
3	0	1	2
2	3	0	1
1	2	3	0

Fig. 5. Latin Square of side 4 on the symbol set $\{0, 1, 2, 3\}$.

is that in any set of N consecutive cells for an output no two reside in the same memory. Note that this set of N consecutive cells need not arrive in consecutive time slots. The set consists of any set of N sequential arrivals for one output spread over an arbitrary period of time. But how does that guarantee that no conflicts occur on the outputs? There are N output ports and the next packet for all of them could well reside in the same memory. However we are guaranteed that the next N packets for each of them are spread across N memories. Even if this set of N memories overlaps completely, there will always be a permutation possible such that only one output reads from one memory at a time. This set of N reads for each output constitutes a scheduling step.

For example in the worst case all outputs want to read from the same set of 4 memories numbered $\{0, 1, 2, 3\}$ in a 4×4 switch. Even in such a case the simple Latin Square of Figure 5 would yield a suitable permutation. Each row of the Latin Square represents a substep and the columns correspond to access by a particular output port. Note that during one substep a memory is accessed only once.

The minimum number of memories needed to realize this scheme is also at most $(2N - 1)$. The set of arrival conflicts is the same as before. However since we are no longer using departure times to ensure conflict resolution on the outputs, so there are no departure conflicts. Instead we have *stripe conflicts* when two packets, within a set of consecutive N packets, are placed in the same memory. The cardinality of the set of stripe conflicts for a given packet is $(N - 1)$. As before, $(2N - 1)$ memories are sufficient.

In subsequent discussion we will assume that each output has a *service scheduler* which runs a work-conserving service scheduling discipline. This service scheduler makes the decision about which flows to read from, and how many packets to read from each. Once that decision has been made it is communicated to the *output controller* which then handles the job of reading these cells from the memories. We want to stress that in the design of our switch we are not concerned with the design of any particular service scheduler. Our objective is only to design an output controller which can read a specified number of packets from any of its flows in a work conserving fashion.

A. Handling multiple flows per output

We will now show how this scheme of striping cells across memories can be used in the case where each output port handles multiple flows. In order to do so we need to determine the size of the stripe, the number of memories needed and the size of the scheduling step. Clearly the number of memories needed

depends on the size of the stripe and the exact relationship is given by the following theorem.

Theorem 2: In a switch with N input ports using a stripe of size P , a set of $M = (N + P - 1)$ memories is sufficient to ensure that cells belonging to any number of flows can be striped across at least P memories.

Proof: As before, the set of arrival conflicts has cardinality $(N - 1)$. The set of stripe conflicts has cardinality $(P - 1)$ because an arriving packet for a given flow cannot be placed in any memory which stores any of the previous $(P - 1)$ packets for this flow. Therefore the total number of conflicts is $(N - 1) + (P - 1)$ and by the pigeon hole principle $M = (N + P - 1)$ memories are needed. Note that this result is independent of the number of flows supported by each output port. This is because there are N input ports and therefore cells belonging to a maximum of N different flows can arrive at a time. ■

The next step is to determine the stripe size P . Here it is helpful to analyze the case where P is equal to N , as we saw that this value of P was sufficient in the case of a FCFS OQ switch. If each output port always reads N packets from the chosen flow the situation is essentially the same as for the FCFS case and the reads can easily be scheduled without conflicts. However the chosen flow could have less than N packets which could lead to loss of throughput. The switch would no longer be work conserving. It is also not possible to wait for more packets to show up because this will also result in a non-work-conserving switch. The obvious solution in such a situation is to read the rest of the packets from another flow. Here we run into a problem. Packets are only guaranteed to be striped within a flow, and it might not always be possible to read from two or more different flows at a time.

If we use a bigger scheduling step (and hence a bigger output memory), it is possible to avoid this problem. The size of the scheduling step will determine the maximum number of cells per flow that can be read at a time if enough cells are available for each flow. Then for a completely fair scheduling discipline (FQ) we would need to read an equal number of cells per flow and this would require a scheduling step of size fN , where f is the number of flows per output. For Weighted Fair queuing (WFQ) the size of the scheduling step would be $(\sum w_i)N$ (assuming integer values for the weights). In order to simplify analysis we will consider the simple FQ case where the scheduling step is of size fN .

The service scheduler hands the output controller a list of flows and the number of cells to be read from each. It is the job of the output controller to create a conflict free set of accesses from this list. The accesses are done N at a time so fN accesses would require f such sub-steps. Within one access list of size N no memory should be listed twice. This would enable the memory controller to permute the access lists of different output controllers to come up with a conflict free memory access pattern.

Note that this scheme still does not guarantee that the switch would be work conserving. In order to appreciate this fully consider a 4×4 switch in which each output handles four different flows. Suppose the output controller for one of the outputs needs to read one cell from three of the flows and the rest from

the fourth one. If each memory operates at the line rate, then this switch would have seven parallel memories. Remember that we only guarantee that cells for one flow are striped across N memories. Therefore in this set of 16 reads the 13 cells for the fourth flow would definitely be spread across 4 memories. It could happen the one cell for each of the other outputs is in, say, the first memory. A maximum of four cells for the fourth flow can be in one memory and it so happens that they are also in the first memory. Therefore seven cells for this output need to be read from one memory. It is the job of the output controller to come up with a four step schedule such that within each step no memory is accessed twice. How can the output controller schedule seven reads from the same memory in four steps? Clearly it cannot be done without adding a speedup. We will call these conflicts *memory read conflicts*.

For our example a speedup of $7/4$ is sufficient to avoid the memory read conflicts. In general, we have the following Speedup Theorem:

Theorem 3: In an $N \times N$ switch with f flows per output, if the packet placement algorithm on the input side ensures that all consecutive packets of a flow are striped across N memories, a scheduling step of size fN on the output side and a speedup of $(2 - \frac{1}{N})$ are sufficient for the output queued switch to support any work conserving scheduling discipline.

Proof: Let F_i be the number of cells to be read from flow f_i , then we know that the maximum number of memory read conflicts for any memory is given by:

$$\begin{aligned} \sum \left\lceil \frac{F_i}{P} \right\rceil &= \sum \left\lfloor \frac{F_i + P - 1}{P} \right\rfloor \\ &= \sum \left\lfloor \frac{F_i - 1}{P} \right\rfloor + f \end{aligned}$$

In particular if $f = N$ and $P = N$ we have,

$$\begin{aligned} \sum \left\lfloor \frac{F_i - 1}{N} \right\rfloor + N &\leq \left\lfloor \frac{\sum (F_i - 1)}{N} \right\rfloor + N \\ &= \left\lfloor \frac{N^2 - N}{N} \right\rfloor + N \\ &= \left\lfloor \frac{N(N - 1)}{N} \right\rfloor + N \\ &= N - 1 + N \\ &= 2N - 1 \end{aligned}$$

Therefore in order to be able to resolve these $(2N - 1)$ memory read conflicts in N scheduling steps we would need a speedup S given by:

$$S = \frac{\text{Number of memory read conflicts}}{\text{number of scheduling steps}}$$

$$S = \frac{2N - 1}{N} = 2 - \frac{1}{N}$$

■

B. Complexity analysis

A memory is compatible with a packet if it does not store any of the previous P packets for that flow. The scheduler still needs to compute a matching. However, no prefix sum calculation is necessary since departure times are not being used. Thus, the scheduler for the FCFS output queued switch can be generalized to work with multiple service classes with a general service discipline on the output ports, while still preserving the $O(\log^2 N)$ time complexity.

VI. IMPLEMENTATION ISSUES

Even though our scheduler is simple relative to most PRAM algorithms, it is not clear that it will outperform a tightly coded sequential implementation for realistic values of N . Ideally, for the fastest possible implementation, we would like to build the scheduler directly in hardware. It is known that by using logic circuits for sorting and arithmetic, basic PRAM operations can be efficiently simulated by logic circuits. Thus, in principle, our PRAM-based parallel algorithm for computing the schedule can be implemented directly by a logic circuit whose depth is $O(\text{polylog}(N))$. In practice, the construction of a logic circuit from a PRAM is relatively inefficient. It is straightforward to map the operations of our scheduler into dedicated hardware units. The only somewhat complicated operation is the 2-coloring procedure, which involves a number of pointer redirections.

VII. CONCLUSION

We have shown that idealized output queuing can be simulated in a shared memory architecture with $3N - 2$ packet memories running at the line rate, using a scheduling algorithm whose time complexity is $O(\log^2 N)$ on a parallel random access machine (PRAM). The number of processing elements and memory cells used by the PRAM are a small multiple of the size of the idealized switch. This scheduler can handle both FCFS and general output queuing. In future we plan to study the practicality of implementing such a scheduler based on the ideas presented in this paper. As part of this effort, we will investigate the possibility of efficiently approximating output queuing.

REFERENCES

- [1] S. Keshav, *An Engineering Approach to Computer Networking*, Addison-Wesley, 1997.
- [2] N. McKeown, "islip: A scheduling algorithm for input-queued switches," *IEEE Transactions on Networking*, vol. 7, no. 2, Apr. 1999.
- [3] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input output queued switch," in *IEEE Infocom*, 1999.
- [4] S. Iyer, A. Awadallah, and N. McKeown, "Analysis of a packet switch with memories running at slower than the line rate," in *IEEE Infocom*, 2000.
- [5] T. H. Cormen, C. E. Leiserson, and R. H. Rivest, *Introduction to Algorithms*, MIT Press, 1989.
- [6] T. Feder, N. Megiddo, and S. Plotkin, "A sublinear parallel algorithm for stable matching," in *Symposium on Discrete Algorithms*, 1994.
- [7] V. Vazirani, *Synthesis of Parallel Algorithms*, chapter Parallel Graph Matching, pp. 783–812, Morgan Kaufmann Publishers, 1993, J. Reif (Editor).
- [8] R. Diestel, *Graph Theory*, Springer Verlag, 2000.
- [9] G. Lev, N. Pippenger, and L. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Transactions on Computers*, vol. 30, no. 2, Feb. 1981.

A. Direct implementation of matching routine

We run $\log \Gamma$ iteration of the 2-coloring based reduction procedure described below. Each iteration of the 2-coloring based reduction procedure reduces the degree of all the vertices to either the ceiling or floor of half its degree prior to the iteration. Since Γ is an integral power of 2, after $\log_2 \Gamma$ iterations we get exactly one edge incident to vertices from the left which started out with a nonzero degree. Since the initial degree of nodes on the right was less than or equal to N , after the $\log_2 \Gamma$ iterations their degree is either 1 or 0, i.e., no 2 vertices on the left are matched to a common vertex on the right. So we get a matching as all the vertices have at most one edge matching them to one of the vertices on the left and all the vertices on the right have at most one vertex on the left matching it.

1) *2-coloring based reduction procedure:* The 2-coloring procedure constructs a new graph with maximum degree 2. So we first group edges going out of a vertex into groups of two and create one vertex per group. If the degree of vertex is odd in the original graph, one of the groups contains only one vertex. We put an edge between two vertices corresponding to two groups in the new graph if and only if the edge belonged to the two groups. So the new graph again is a bipartite graph with maximum degree 2. Since the maximum degree of the graph is 2, it will consist of disjoint cycles and chains only and because it is a bipartite graph it can contain cycles of even length. So we can color the edges of this graph in 2 colors in $\log N$ time by initializing every edge as a chain of size one and color zero and after that at each step merging adjacent chains into a single chain by making their color consistent. Details can be found in [9]. After this step we throw away all the edges in the original graph corresponding to edges with color 1 in the new graph. This reduces the degree of each node to either the ceiling or floor of its half.

Since each 2-coloring iteration takes $O(\log N)$ time, the whole matching procedure takes $O(\log^2 N)$ time and the number of vertices in the later graph is $O(N^2)$, we need $O(N^2)$ processors.

2) *Complexity analysis:* In the 2-coloring algorithm we get a node for every edge in the original graph so there will be $O(N^2)$ nodes with maximum degree 2. Using pointer doubling 2-coloring can be done in $O(\log(N^2)) = O(\log N)$ time using $O(N^2)$ processors. There is also a nontrivial amount of work in pairing the edges in each step. This again can be done by doing a prefix sum operation at each node in $O(\log N)$ time using $O(N)$ processors at each node. So a total of $O(N^2)$ processors are required in order to do each iteration of the 2 phase coloring in $O(\log N)$ time. Since we need to do $O(\log N)$ iterations, the total time complexity is $O(\log^2 N)$. In each step we need to store a constant amount of information per edge so the memory required for this operation is again $O(N^2)$. Hence all the work of the scheduler can be done using $O(N^2)$ processors and $O(N^2)$ memory in $O(\log^2 N)$ time.