Deficit Round-Robin Scheduling for Input-Queued Switches

Xiao Zhang, *Student Member, IEEE,* and Laxmi N. Bhuyan, *Fellow, IEEE*

***Abstract*—In this paper, we address the problem of fair scheduling of packets in Internet routers with input-queued switches. The goal is to ensure that packets of different flows leave a router in proportion to their reservations under heavy traffic. First, we examine the problem when fair queuing is applied only at output link of a router, and verify that this approach is ineffective. Second, we propose a flow-based iterative deficit-round-robin (iDRR) fair scheduling algorithm for the crossbar switch that supports fair bandwidth distribution among flows, and achieves asymptotically 100% throughput under uniform traffic. Since the flow-based algorithm is hard to implement in hardware, we finally propose a port-based version of iDRR (called iPDRR) and describe its hardware implementation.**

***Index Terms*—Fair scheduling, input-queued crossbar switch, quality-of-service (QoS).**

I. Introduction

MOST OF THE commercial routers employ input-queued switches (Cisco [1], BBN [2]) because output-queued switches are difficult to design. Fairness in resource allocation is important to support quality-of-service (QoS). In this paper, we address the problem of fair scheduling of packets in routers with input-queued switches. The goal is to make sure that packets of different flows leave a router in proportion to their reservations when the bandwidth of the router cannot accommodate all incoming traffic.

Generally, a router consists of line cards which contain input and output ports, a switch fabric, and forwarding engines. When a packet arrives at an input port, its header is sent to the forwarding engine, which returns the output port and the new header. The packet is buffered in the input queues waiting for the switch to transfer it across the crossbar switch fabric. When the packet is received at the output port, it is pushed into the output queues. The packet leaves the router when the output link is ready.

Many fair queuing algorithms [3]–[7] have been developed to schedule packets on shared resources. Some of these have been implemented at the output links of commercial routers [1], [2] to support QoS. However, with today's technology, the link speed is increasing almost exponentially. The backbone crossbar with its scheduler becomes the real bottleneck, and most of the packets are waiting at the input buffer instead of the output buffer. Therefore, applying fair scheduling at the crossbar switch should be more effective than that at the output queue. Our simulation results in this paper confirm this hypothesis.

We propose a flow-based iterative deficit-round-robin fair scheduling algorithm (iDRR) for the crossbar switch. Our experiments shows that it provides high throughput, low latency and fair bandwidth allocation among competing flows. The algorithm is based on the deficit round-robin (DRR) algorithm [6]. Another flow-based crossbar scheduling algorithm, called iterative fair scheduling (iFS), was proposed in [8]. It is based on virtual time [4], [5]. We believe that the DRR algorithm is easier to implement in hardware than the virtual time. We also show that iDRR delivers packets at a rate close to an output-queued switch.

In practice, if an algorithm has to be fast, it is important that it be simple and easy to implement in hardware. A flow-based fair scheduling algorithm is desirable, but is difficult in terms of hardware implementation because of the large and variable number of flows. To solve this problem, we also develop a port-based iterative deficit-round-robin fair scheduling algorithm (iPDRR) that can work with VOQs [9]. We propose to divide the flow-based scheduling into two stages. First, a fair queuing algorithm is applied at the input buffer to resolve contentions among flows from same inputs to same outputs. Then, a port-based fair scheduling algorithm is adopted to resolve contentions among ports.

Stiliadis and Varma also proposed a port-based fair scheduling algorithm weight probability iterative matching (WPIM) [10]. It was shown that iFS performs better than WPIM in terms of granularity of fairness [8].

The rest of the paper is organized as follows. In Section II, we describe the architecture of the input-queued switch used in this paper. In Section III, we examine the problem when fair queuing is applied only at the output link. Then, we propose a flow-based iterative deficit-round-robin fair scheduling algorithm iDRR for the switch in Section IV. In Section V, we present a port-based algorithm iPDRR. Its hardware implementation is described in Section VI. Finally, Section VII concludes this paper.

II. Operation of the Input-Queued Switch

Fig. 1 shows the internal structure of the input-queued backbone switch of a router. We explain the operations of various components of this figure in this section. Incoming packets are stored in the input queue, which has a separate queue per output port, called virtual output queue (VOQ), if a port-based switch scheduler is used. If the switch scheduler is flow-based, a separate queue per flow needs to be maintained. We call it virtual flow queue (VFQ).

Manuscript received August 1, 2002; revised January 16, 2003. This paper was supported in part by the National Science Foundation (NSF) under Grant CCR 0105676.

The authors are with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA (email: xzhang@cs.ucr.edu; bhuyan@cs.ucr.edu).

Digital Object Identifier 10.1109/JSAC.2003.810495

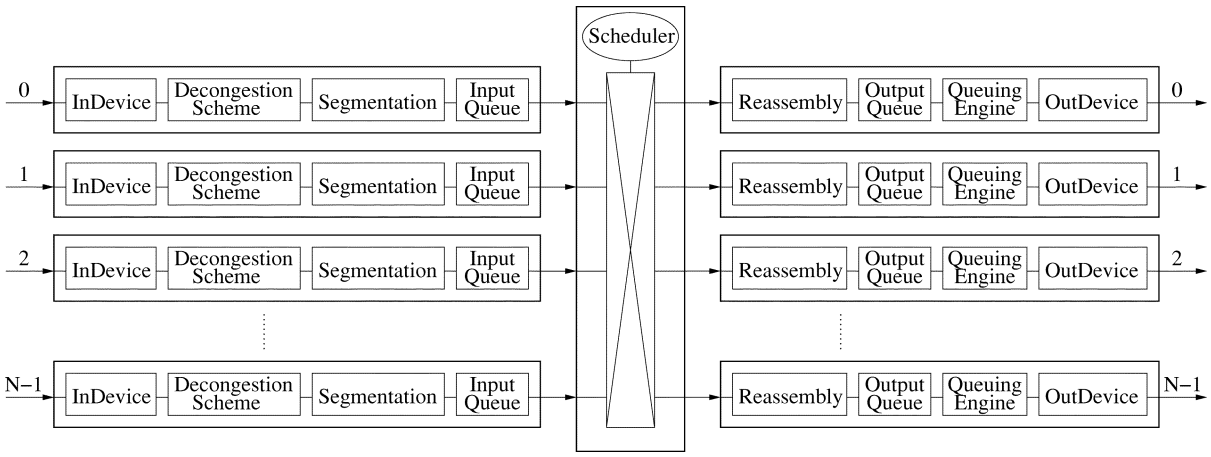

Fig. 1. Block diagram of an input-queued switch.

The size of the input buffer is finite. When the input buffer is full or congestion is anticipated, a decongestion mechanism is needed to determine when to drop packets and which packets to drop. One function of the decongestion mechanism is to isolate bad-behaved flows from well-behaved ones. Such flow isolation is critical to fair scheduling [8]. In the rest of this paper, we assume that such a mechanism is already provided.

The center of the router is the switch fabric, usually an $N\times N$ crossbar connecting N inputs and N outputs. It operates on small fixed-size units, or *cells* (in ATM terminology). A slot is the time to transfer one cell across the switch fabric. Variable-length packets are usually segmented into cells before being transferred across the crossbar. The switch scheduler selects a configuration of the crossbar to ensure that at any instance, each input is connected to at most one output, and each output is connected to at most one input. For each pair of matched input and output, a cell is selected to be transferred across the switch. In practice, the switch scheduling functionality is distributed to each port, and an iterative scheduling algorithm is implemented to work independently for each input and output [8], [10]–[16]. We modify this part of the hardware to implement fair scheduling.

The scheduling can be done at the cell level, i.e., all the input–output connections are torn down after each slot and the scheduler considers all inputs and outputs to find a new matching. Cells from different packets can be interleaved to one output. This is the scheme used in many current routers.

Alternatively, the scheduling can be done in such a way that when an input and an output are matched, the connection is kept until a complete packet is received at the output. This approach is called *packet-mode scheduling* while the previous is called *cell-mode scheduling*. It was shown that packet-mode scheduling has better packet delay performance than cell-mode scheduling in case of packet length distribution with a small variance (the coefficient of variation of the service time is less than one) [17].

The reassembly module holds cells until the last cell of a packet is received at the output port, and then reassembles these cells back into a complete packet. In cell-mode scheduling, N reassembly engines are required, where N is the number of inputs if port-based scheduling is used or the number of flows if flow-based scheduling is adopted. In packet-mode scheduling, only one reassembly engine is needed.

When a complete packet is received by the reassembly module, it is stored in the output queue. The output queue can be a simple first-in-first-out (FIFO) queue if the basic first-come-first-served (FCFS) queuing algorithm is used, or a multiple-queue buffer if some fair queuing algorithm, such as DRR, is employed. When a output buffer fills up, the switch is notified to stop transferring packets to it until some buffer space frees up.

The output link is responsible for sending packets out of the router. When the output link is ready, it signals the output queuing engine to pick a packet from the output queue according to a certain criterion. Commercial routers, such as Cisco 12000 series [1], implement DRR-based algorithms at this point.

Our simulation is based on the model shown in Fig. 1. Packet arrival is modeled as a two-state ON-OFF process. The number of ON state slots is defined as the packet length which is generated from a profile of NLANR trace at AIX site [18]. We collected 119,298,399 packets from Sunday, May 12, 02:45:06, 2002 to Saturday, May 18, 23:11:34, 2002. The packet length ranges from 20 to 1500 bytes with a mean $(E_{\rm on})$ of 566 bytes and a standard deviation of 615 bytes. The number of OFF state slots is exponentially distributed with average $E_{\rm off} = ((1-p)/p)E_{\rm on}$, where p is defined as offered workload $(0 < p < 1)$. Cell size is 64 bytes. Input buffer size is 10240 cells. We have varied these parameters to do a sensitivity analysis. However, due to page limit, we report only a few results in this paper based on above parameters.

III. Problem With Fair Queuing at the Output Link

Fair queuing is usually applied at the output ports of a router. In this section, we verify that this approach is ineffective in a router with input-queued switch.

Consider a 4×4 IP router configured as Fig. 1. We choose iSLIP [12] as the switch scheduler and DRR [6] as the fair queuing algorithm. Each input has one flow destined to output 0, and reserves 10%, 20%, 30%, and 40% bandwidth, respectively. Each flow maintains the same arrival rate. Fig. 2 shows

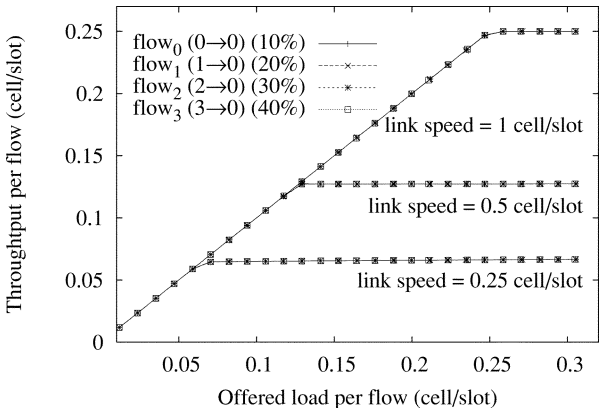

Fig. 2. Throughput per flow using iSLIP and DRR.

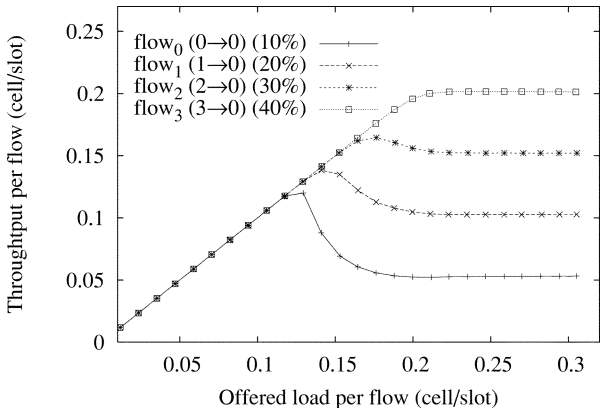

Fig. 3. Throughput per flow using iSLIP and DRR. ESPF is applied right before the output queue. Output-link speed $= 0.5$ cell/slot.

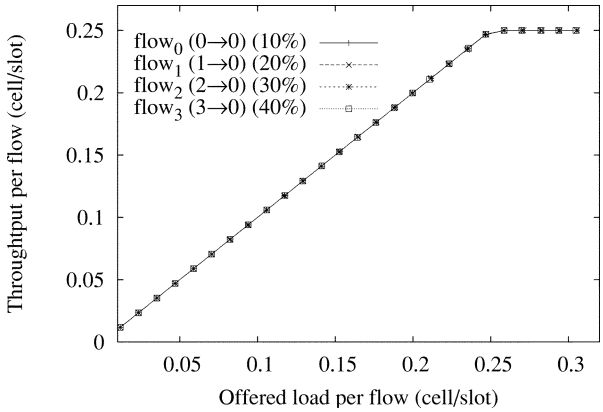

Fig. 4. Throughput per flow using iSLIP and DRR. ESPF is applied right before the output queue. Output-link speed $= 1$ cell/slot.

the throughput of each flow when the output link speeds are 1, 0.5, and 0.25 cell/slot, respectively.

In the case of 1 cell/slot link speed, DRR cannot distinguish among flows. It is because DRR works only if there are backlogged packets in the queue. But the number of packets in the output queue is very limited because the output link speed is the same as that of the switch.

When the link speed is less than 1 cell/slot, DRR also fails to distinguish among flows. The underlying problem is that the output queue lacks a decongestion mechanism to isolate different flows. The switch pushes packets of different flows into the output queue at the same rate. On the other hand, DRR pops packets of different flows in proportion to their reservations. Eventually, most of the output buffer space is filled up by flows with the lowest reservation.

However, applying a decongestion mechanism at the output queue has its own problem. Suppose that we apply a decongestion scheme called equal size per flow (ESPF)[1] [8] right before the output queue, one solution could be: when a flow uses up its quota, the switch is informed to stop transferring packets belonging to that flow. However, this approach will not work since iSLIP does not distinguish among flows. Another way is to drop the packet when a flow uses up its quota. This approach works (as shown in Fig. 3), but is still not very attractive. First, accepting packets at input queue and dropping them at output queue wastes switch bandwidth and input buffer space. Second, applying a decongestion mechanism at the output side is redundant since it is already done at the input side. Finally, this approach still cannot solve the problem when output-link speed is the same as the switch speed (see Fig. 4).

The conclusion is generally true. If the switch scheduler treats flows with different reservations equally, it is useless to apply fair queuing only at the output side. In other words, the fairness issue should be addressed at the switch and/or input side. In the following section, we develop such a flow-based scheduler for the switch and use simple FIFO output queues.

[1]ESPF: Each flow is assigned a quota which equals the total buffer size divided by the number of flows. When the buffer occupancy is below a certain threshold, all incoming packets are accepted. After that, a packet is accepted only if the flow's quota has not been used up.

IV. A Flow-Based Fair Scheduling Algorithm

Based on the observation in Section III, a straightforward idea is to apply fair queuing at the switch to support fair bandwidth allocation. In this section, we first introduce the definition of fairness in input-queued switch scheduling, and then present an iterative deficit-round-robin fair scheduling scheme. We call it iDRR.

A. Definition of Fair Scheduling

We follow the same definition of fairness as that in [8]. An input-queued switch is work conserving. Let r_k be the reservation of flow$_k$, and $W_k(t_1, t_2]$ be the amount of flow$_k$ traffic served in the interval $(t_1, t_2]$. For any two backlogged flows f_i and f_j that are in contention, a scheduling scheme is fair in $(t_1, t_2]$ if

$$\frac{W_i(t_1, t_2]}{W_j(t_1, t_2]} \geq \frac{r_i}{r_j}.$$

Intuitively, this definition means that when the bandwidth of a router cannot accommodate all incoming packets, packets of different flows leave the router in proportion to their reservations.

In a router with input-queued switch, the input/output line cards and the switch are usually of the same speed. In this scenario, the switch input port cannot be overloaded because the incoming traffic has been shaped by the input line card which

receives packets at the rate of 1 cell/slot. The switch output port, however, can be overloaded when packets from different input ports go to the same output port, or the instantaneous output line speed is reduced because of the slowdown of the down link. These are some of the "heavy traffic" situations we consider in this paper.

B. Description of iDRR Algorithm

The basic idea of iDRR is to assign each flow a quota which is in proportion to its reservation. When a flow's corresponding input and output are matched, we continue transferring packets of the flow until its quota is used up.

In an $N \times N$ switch, for each flow_k, we maintain a separate queue VFQ_k and three values: quota_k, counter_k and candidate_k. The value of quota_k is in proportion to the flow_k's reservation (see Section IV-C1). In packet-mode scheduling, we assume that the maximum transfer unit (MTU) is known in advance, and the minimum quota is no smaller than MTU (This assumption is not necessary in cell-mode scheduling). Counter_k indicates the current available quota of flow_k. At each $\text{input}_i (0 \leq i \leq N-1)$, a linked list inflowList_i is maintained to record the active flows from input_i. At each $\text{output}_j (0 \leq j \leq N-1)$, another linked list outflowList_j is maintained to record active flows to output_j. When flow_k (from input_i to output_j) becomes active (VFQ_k was empty and a packet of flow_k arrives), add it to the end of inflowList_i and outflowList_j, and initialize counter_k to -1 and candidate_k to *FALSE*. When flow_k becomes inactive (VFQ_k becomes empty for a period of time after a packet of flow_k leaves, see Section IV-C2), it is removed from inflowList_i and outflowList_j. Initially, all inputs and outputs are unmatched. Then, in each iteration:

1) *Request*: Each unmatched input_i sends a request to every output for which it has a queued cell.
2) *Grant*: If an unmatched output_j receives any requests, choose from outflowList_j the first flow, say flow_k (from input_i to output_j), such that input_i sends a request and VFQ_k is not empty. Set candidate_k to *TRUE* and send a grant to input_i.
3) *Accept*: If an unmatched input_i receives any grants, choose from inflowList_i the first flow, say flow_k (from input_i to output_j), such that $\text{candidate}_k = \text{TRUE}$. Increase counter_k by quota_k. Move flow_k to the end of inflowList_i and outflowList_j. Set candidate_l to *FALSE* for all flow_l in inflowList_i. Send an accept to output_j.

   Input_i and output_j are then set as matched. And flow_k is marked as the selected flow for the $\text{input}_i - \text{output}_j$ pair.

In each slot, for every selected flow_k, the switch transfers a cell of its head-of-line (HOL) packet and decrease counter_k by 1. After transferring a complete packet (if packet-mode scheduling is used) or a cell (if cell-mode scheduling is used), if $\text{counter}_k < 0$ or VFQ_k is empty, reset input_i and output_j as unmatched.

C. Remarks

1) Reservation and Quota: For any flows wishing to receive guaranteed bandwidth, reservation is necessary. When a flow makes a reservation at a router, its identification and reservation information is entered into the flow list of the router. All packets of this flow will carry the identification information.

Flow reservation can be made as the percentage of the total bandwidth. The quota can be set statically or dynamically. In the static approach, we assign a minimum quota $(q_{\min})$, say 24 cells, to the minimum available reservation $(r_{\min})$. Then, for a flow with reservation r, its quota is $(r/r_{\min})q_{\min}$. In the dynamic approach, the minimum quota is assigned to the flows with minimum reservation and quotas of other flows can be adjusted accordingly.

2) Flow Deactivation: After a flow makes a reservation, it can be active or inactive depending on its queue status until the reservation is canceled. Initially VFQ_k is empty and flow_k is inactive. When the first packet of flow_k arrives, flow_k becomes active. When VFQ_k becomes empty, the first thought is to deactivate flow_k immediately. However, doing so may cause some problems. Let us see the following two cases.

- Suppose packet-mode scheduling is used, and the length of each packet of flow_k is 20 cells. At one moment, counter_k is 1 and flow_k is serviced. After one packet is transferred, counter_k becomes -19 and VFQ_k becomes empty. If flow_k is deactivated immediately, the over-used counter (-19) will be cleared. Then, if flow_k becomes active after one slot, it may over-use its reserved bandwidth.
- Likewise, immediate deactivation may also cause under-use of a flow's reservation. Suppose each packet of flow_k is 1 cell long. At one moment, counter_k is 100 and flow_k is serviced. After one packet is transferred, counter_k becomes 99 and VFQ_k becomes empty. Deactivating flow_k immediately will wipe out the unused counter (99).

Therefore, we propose that *a flow becomes inactive only after its VFQ becomes empty for a period of time*. The timeout value of an active flow after its queue is empty can be set statically, say 1000 slots. If no more packets arrives within this period of time, the flow is considered inactive.

Alternatively, we can deactivate a flow dynamically as follows. Let r_c be the current total reservation of all flows competing with flow_k of reservation r_k. When VFQ_k becomes empty and $\text{count}_k \neq -1$, decrease (if $\text{counter}_k \geq 0$) or increase (if $\text{counter}_k < -1$) counter_k by 1 every $(r_c + r_k)/(r_k)$ slots until $\text{counter}_k = -1$. Then, set flow_k as inactive and remove it from the two active flow lists.

When the above flow deactivation scheme is used, it may so happen that a flow is in the active flow list while its queue is empty. Therefore, in the grant step of iDRR, when selecting a flow, the condition "VFQ_k is not empty" is necessary.

3) Complexity: In the context of fair queuing, assuming that MTU is known in advance and letting all quotas no smaller than MTU ensure that DRR has the time complexity of $O(1)$, i.e., it is guaranteed that the top of the active flow list can be selected. In [19], Kanhere *et al.* proposed an elastic round-robin (ERR) algorithm which removes this assumption by partitioning the time into rounds and using the maximum packet length during the previous round as the minimum quota in the current round. In the context of switch scheduling, however, $O(1)$ still cannot be guaranteed even with the above modifications. Because it

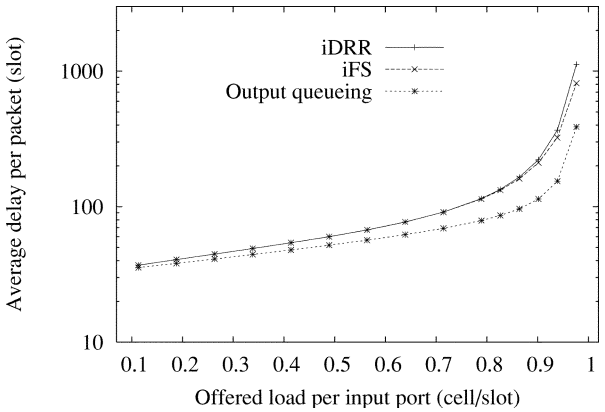

Fig. 5. Performance of iDRR, iFS, and output queuing under uniform traffic.

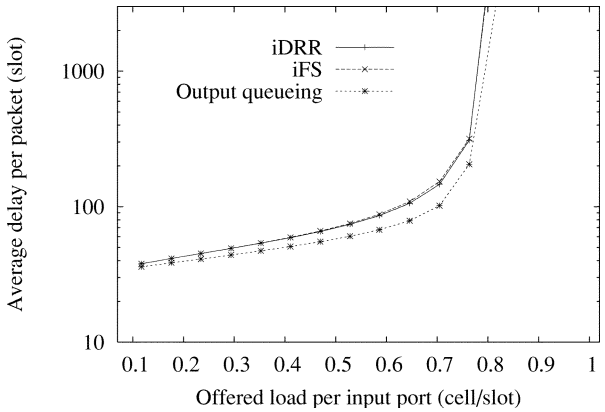

Fig. 6. Performance of iDRR, iFS, and output queuing under nonuniform traffic.

may happen that the input port of the top flow in the *outflowList* has already been matched in the previous slots. In the worst case, the algorithm has to check all flows in the list before it selects one.

4) Cell Mode Versus Packet Mode: Scheduling can be cell mode or packet mode. In the case of flow based scheduling, cell mode may not be a good choice. Although cell-mode scheduling simplifies the switch, it requires that the reassembly module can hold N packets at one time, where N is the number of flows and can be very large. Therefore, we choose packet mode in our simulation.

D. Simulation Results

Fig. 5 shows the throughput and delay performance of iDRR compared with another flow-based scheduling algorithm *iFS* under uniform traffic. The performance of output queuing with DRR applied at the output queue is also shown for comparison. The simulation is run on a 16×16 switch. Each input has two flows to each output, totally 512 flows. Each flow reserves the same bandwidth and maintains the same arrival rate. The output-link speed is 1 cell/slot. The number of iterations is three (same as that in Tiny Tera [20]). Under this circumstance, we observe that the average packet delay of iDRR is almost identical to that of iFS. They are all close to the delay when output queuing is used. Hence, iDRR is capable of achieving asymptotically 100% throughput for uniform traffic.

For the nonuniform traffic, we consider the server-client model as used in [8], [10], and [11]. In a 16×16 switch, four ports are connected to servers and twelve to clients. Each client sends 10% of its generated traffic to each server, and the remainder is evenly distributed among other clients. For each server, 95% of its traffic goes to clients, and 5% to other servers. In this setting, the traffic from clients to servers is almost twice that from clients to clients. Fig. 6 shows the average packet delay of traffic from clients to servers as a function of the workload per input. As we can see, iDRR and iFS are almost indistinguishable and can reach a throughput of about 78%.

To evaluate the fairness of iDRR, we run the simulation on a 4×4 switch, where each input has two flows destined to output 0 with different reservations. Each flow maintains the same arrival rate. Output link speed is 1 cell/slot. As shown Fig. 7, the

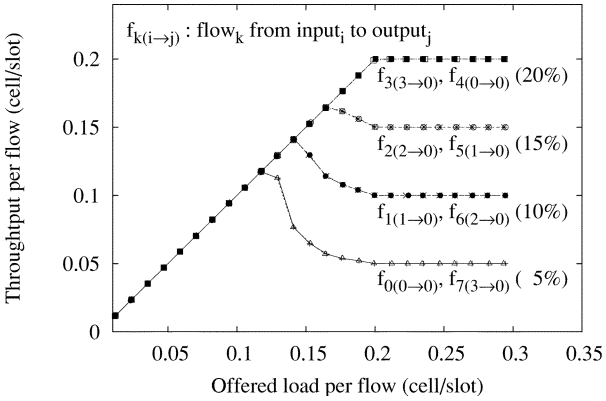

Fig. 7. Throughput per flow using iDRR.

bandwidth is distributed in proportion to each flow's reservation when the link is over-subscribed.

V. Port-Based Fair Scheduling Algorithm

Flow-based fair scheduling algorithms are desirable in terms of fairness among flows. However, in terms of hardware implementation, they are more complex than port-based algorithms. In iDRR and iFS, each port needs to maintain an active flow list, whose length varies from time to time and can be very large.

We propose to divide the flow-based scheduling problem into two stages. Fig. 8 shows two additional stages, VFQ and DRR, introduced at the input side of Fig. 1. There is no fair queuing engine at the output side. First, we apply a fair queuing algorithm at the input buffer to resolve contentions among flows from same inputs to same outputs. The VFQ is implemented in DRAM and the DRR fair queuing is implemented in software. A separate input queue (VOQ) is maintained for each output, as required for iterative scheduling algorithms. Then, we develop a port-based fair scheduling algorithm to resolve contentions among the input ports.

For a hardware scheduling algorithm to be useful, it is important that it be simple. That why iSLIP is the choice in Tiny Tera [20] and Cisco GSR [1], although it does not offer the best performance compared to other schemes or provide 100% throughput under nonuniform traffic [21]. It is readily implemented in hardware and can operate at high speed. It was shown

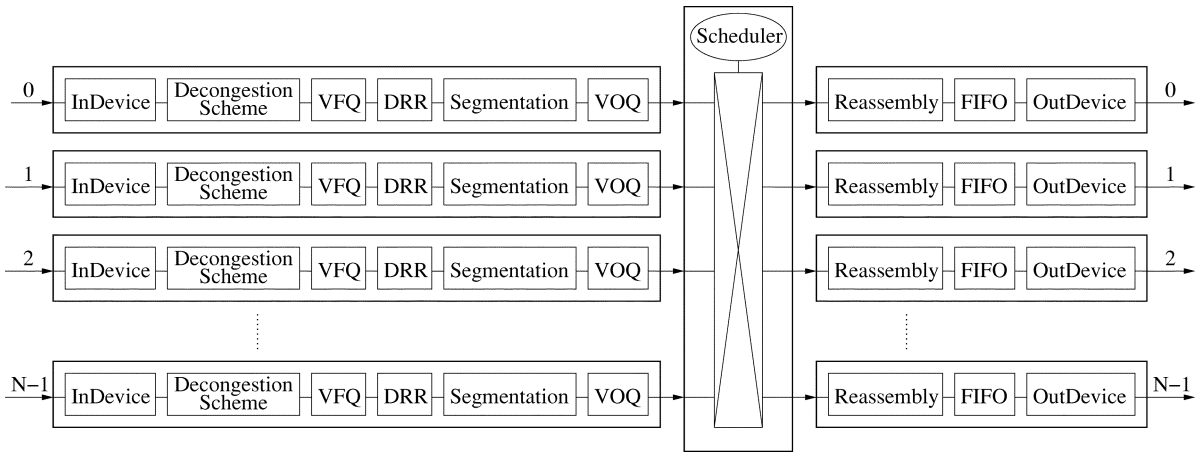

Fig. 8. Two-level scheduling scheme for an input-queued switche.

that iSLIP can find a matching using three iterations within eight switch cycles (45 ns) for a 32×32 switch [22].

Our iDRR can be readily modified to its port-based version. We call it iPDRR. Because of the fixed number of ports, it is easy to implement in hardware. In the rest of this section, we describe iPDRR, compare it with other schemes, and show our simulation results. Its hardware implementation will be discussed in details in the next section.

A. Description of iPDRR

In an $N\times N$ switch, for each pair of input_i–output_j, we maintain a separate queue VOQ_{ij} and two counters: quota_{ij} and counter_{ij}. Quota_{ij} indicates the reserved bandwidth from input_i to output_j. Its value is in proportion to the aggregate reservation of all flows from input_i to output_j. In packet-mode scheduling, we assume that MTU is known in advance, and all quotas are no smaller than MTU (this assumption is not necessary in cell-mode scheduling). Counter_{ij} indicates the current available quota of the $\text{input}_i - \text{output}_j$ pair, initialized to -1. For each output_j, we maintain a fixed-size linked list inportList_j, initialized to $\{0,1,2,\ldots,N-1\}$. For each input_i, we keep another fixed-size linked list outportList_i, initialized to $\{0,1,2,\ldots,N-1\}$. Initially, all inputs and outputs are unmatched. Then, in each iteration:

1) *Request*: Each unmatched input_i sends a request to output_j if VOQ_{ij} is not empty.
2) *Grant*: If an unmatched output_j receives any requests, choose from inportList_j the first i such that input_i sends a request to output_j, and send a grant to input_i.
3) *Accept*: If an unmatched input_i receives any grants, choose from outportList_i the first j such that output_j sends a grant to input_i, and send an accept to output_j.

   Input_i and output_j are now considered matched. Then, increase counter_{ij} by quota_{ij}. Move i to the end of inportList_j, and j to the end of outportList_i.

If input_i and output_j are matched, the switch transfers the HOL packet from input_i to output_j and decreases counter_{ij} by one every slot. In packet-mode (or cell-mode) scheduling, after transferring a complete packet (or a cell), if $\text{counter}_{ij} < 0$ or VOQ_{ij} is empty, set input_i and output_j as unmatched and tear down their connection. Otherwise, keep the connection and continue transferring packets (or cells).

Notice that iPDRR differs from iDRR in that inactive ports are not removed from the linked list so that the size of the linked list is fixed. This modification makes hardware implementation much easier (see Section VI).

Like iDRR, when VOQ_{ij} becomes empty, count_{ij} should be adjusted to prevent users from overutilizing or underutilizing the bandwidth (see Section IV-C2). Let r_{ij} be the reservation of the input_i–output_j pair, and r_c be the total reservation of all input–output pairs competing with $\text{input}_i - \text{output}_j$. When VOQ_{ij} becomes empty and $\text{counter}_{ij} \neq -1$, for every $(r_c + r_{ij})/(r_{ij})$ slots, decrease (if $\text{counter}_{ij} \geq 0$) or increase (if $\text{counter}_{ij} < -1$) by one until $\text{counter}_{ij} = -1$.

B. iPDRR Versus iSLIP

In [12], McKeown proposed a round-robin algorithm iSLIP. Instead of using a linked list, iSLIP uses a pointer at each output(input) to record the input(output) with the highest priority. Grant(Accept) is given in the order starting from the highest priority port.

A weighted iSLIP algorithm is also proposed in [12]. For each output_j, instead of maintaining an ordered circular list $S_j = \{0,\ldots,N-1\}$ as that in the basic iSLIP algorithm, the weighted iSLIP expands the list to $S_j = \{0,\ldots,W_j-1\}$ and i appears $(r_{ij})/(r_j)W_j$ times in S_j, where $W_j = (N \cdot r_j)/(\text{LargestCommonFactor}(r_{ij}))$, r_{ij} is the reservation for the input_i–output_j pair, and r_j is the aggregate reservation for output_j.

Unfortunately, weighted iSLIP cannot readily take the advantage of the basic iSLIP hardware implementation because of the variable size of S_j. When adding or removing a reservation, S_j will be recalculated. The hardware implementation of iPDRR, on the other hand, is very straightforward and easy.

C. iPDRR Versus iPFS

In [8], Ni and Bhuyan proposed a switch scheduling algorithm, called iFS, which is based on virtual time. Each incoming packet is assigned a virtual time according to its flow's reservation. The iFS schedules packets in the increasing order of the virtual time.

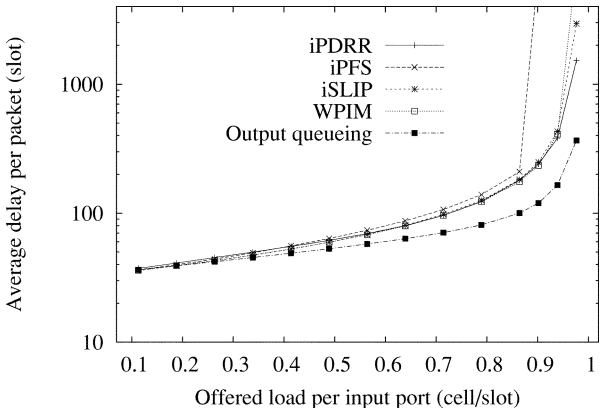

Fig. 9. Performance of iPDRR, iPFS, WPIM, and iSLIP under uniform traffic (cell-mode scheduling).

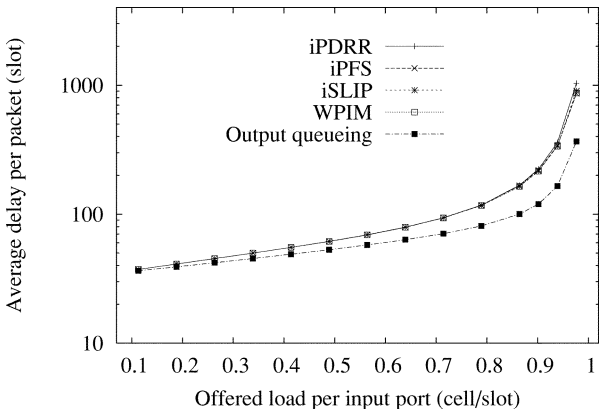

Fig. 10. Performance of iPDRR, iPFS, WPIM, and iSLIP under uniform traffic (packet-mode scheduling).

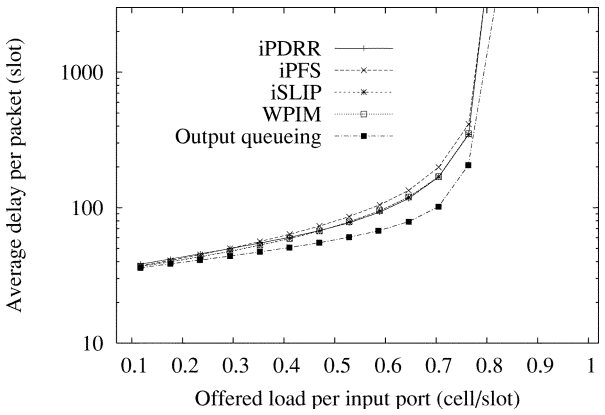

Fig. 11. Performance of iPDRR, iPFS, WPIM, and iSLIP under nonuniform traffic (cell-mode scheduling).

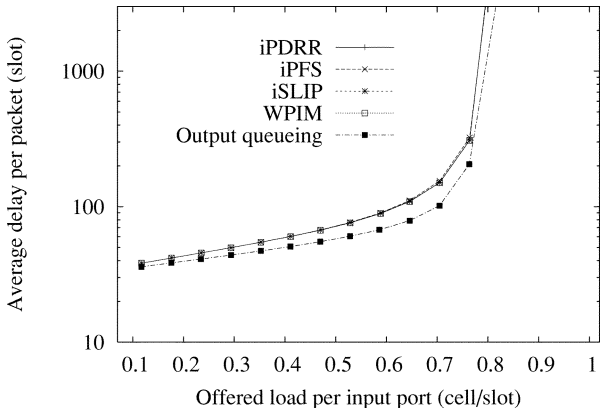

Fig. 12. Performance of iPDRR, iPFS, WPIM, and iSLIP under nonuniform traffic (packet-mode scheduling).

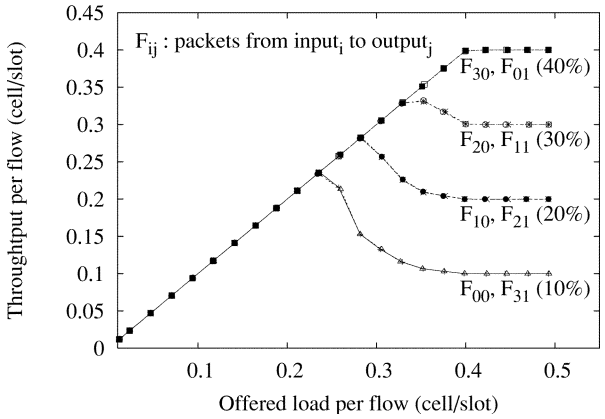

Fig. 13. Throughput per flow using iPDRR (output link speed $=$ 1 cell/slot, packet-mode scheduling).

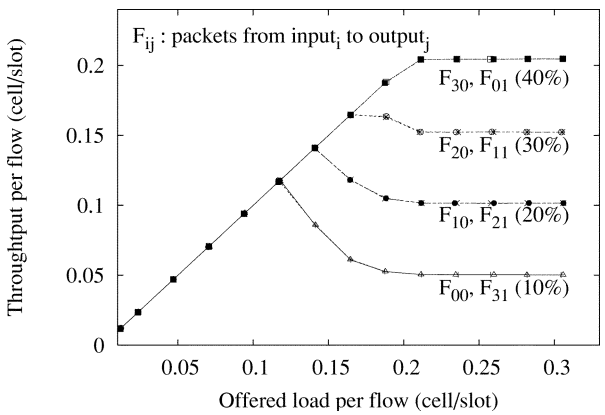

Fig. 14. Throughput per flow using iPDRR (output link speed $=$ 0.5 cell/slot, cell-mode scheduling).

The original iFS is flow based. The time complexity of a schedule arbiter is $O(n)$, where n is the number of flows. If its port-based version is used (we call it iPFS), n is the number of ports and $O(1)$ can be achieved by using parallel comparison. Still we need to compare N values and select the smallest in a very short time. For example, the scheduler of Tiny Tera [20] runs at a clock speed of 175 MHz, and a slot is composed of nine cycles in which eight cycles are for three iSLIP iterations and one cycle for crossbar configuration. If iFS uses such timing, the comparison of N values has to be done within about 11 ns.

Fast comparison prefers small register width. However, calculating virtual time may involve floating point which increases the register width. In addition, virtual time is monotonically increasing. The registers must be big enough to hold the flow of the longest life. A 16-bit register implies that a flow can only live as long as 65535 slots (about 3.4 ms in a switch like Tiny Tera).

iPDRR, as we can see later in Section VI, is easily implemented in hardware and can operate at high speed. Selecting a port from a list can be done by using a circular linked list and

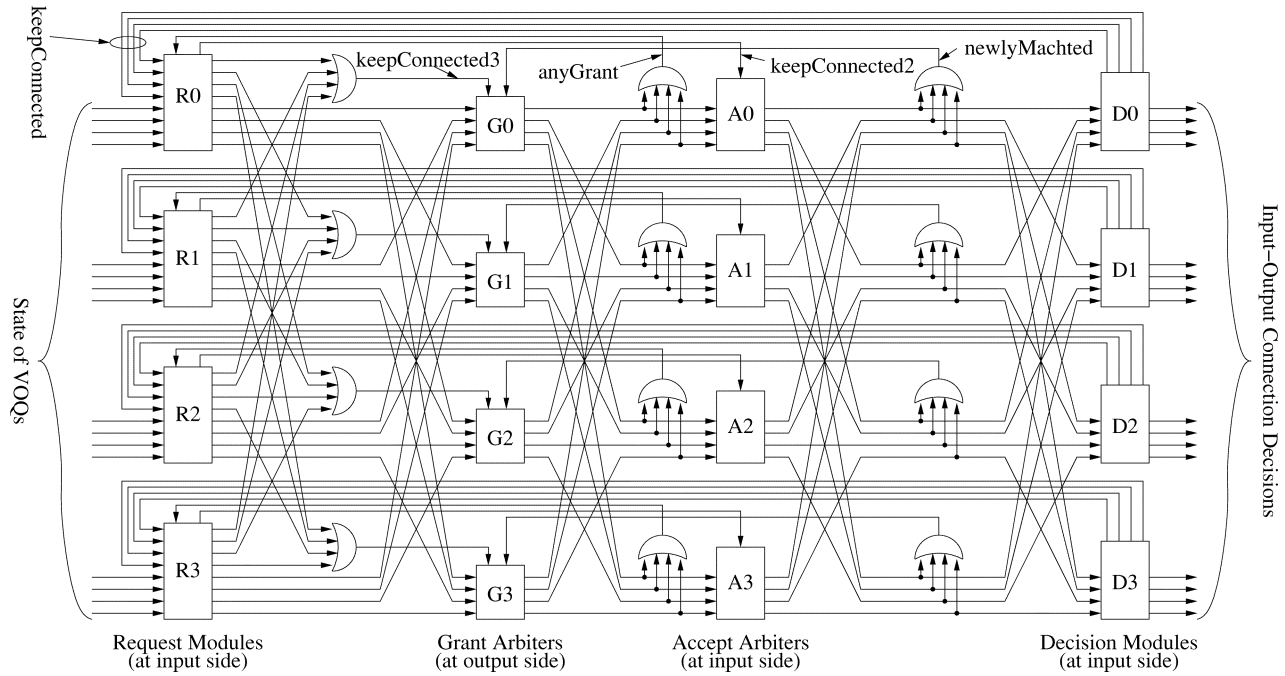

Fig. 15. iPDRR scheduler for a 4 × 4 switch.

a simple combinational circuit with multiplexers and demultiplexers. And connection-tear-down logic can be carried out with $2N$ registers and an adder. All modules in iPDRR can be accomplished in one or two cycles. In addition, a 16-bit register is big enough to hold a flow's quota.

D. iPDRR Versus WPIM

In [10], Stiliadis and Varma also proposed a port-based fair scheduling algorithm WPIM which is based on the original parallel iterative matching (PIM) algorithm [11]. In WPIM, the time axis is divided into frames with a fixed number of slots per frame. The reservation unit is slot/frame. In the first iteration, for each output, an additional mask stage is introduced to block those inputs whose credits are used up, thus allowing other inputs to receive their shares of the bandwidth. Clearly, the bandwidth guarantee is provided at coarse granularity of a frame [8].

The WPIM scheduler uses random selection which is an expensive operation, particularly for a variable number of input requests. It is hard to implement in very short time. In [11], a slot time is 420 ns for four PIM iterations.

WPIM has to be aware of the output link speed, i.e., when the output-link speed changes, the credit of each flow should be changed accordingly. For example, assume that the output-link speed is 1 cell/slot, a frame is 1000 slots, and four flows reserve 10%, 20%, 30%, and 40% bandwidth, respectively, i.e., 100, 200, 300, and 400 slots/frame. If the output-link speed is reduced to 0.5 cell/slot, they will get 10%, 13.3%, 13.3%, and 13.3% bandwidth, respectively, unless the credits of each flow are changed to 50, 100, 150, and 200 slots/frame, respectively. The iPDRR, on the other hand, can always provide fair sharing regardless of the change of the output link speed.

When the actual reservation to an output port is less than its capacity, WPIM equally allocates the rest of the bandwidth among all competing flows. In iPDRR, the rest of the bandwidth is still allocated in proportion to competing flows' reservations.

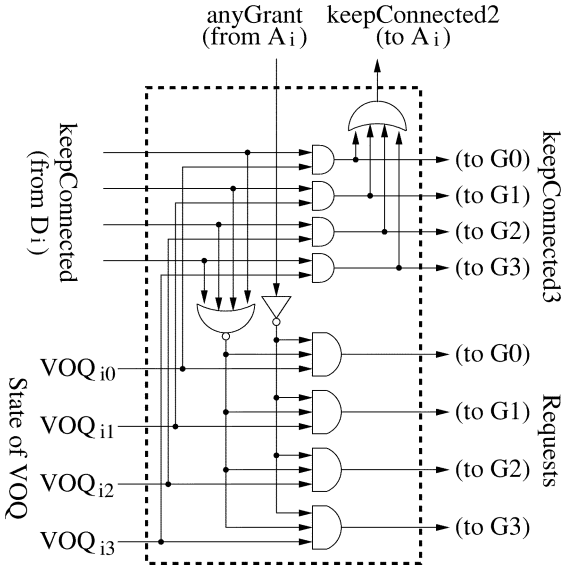

Fig. 16. iPDRR request module R_i at input_i for a 4 × 4 switch.

E. Simulation Results

Fig. 9 and Fig. 10 show the throughput and delay performance of iPDRR compared with other schemes under uniform traffic. The simulation is performed on a 16×16 switch, where each input has one flow to each output, totally 256 flows. Each flow reserves the same bandwidth and maintains the same arrival rate. The number of iterations is three. The output-link speed is 1 cell/slot.

We observe that packet mode can reduce the average packet delay. In packet mode, all schemes are almost the same, while in cell mode, iPDRR works better than other schemes under heavy

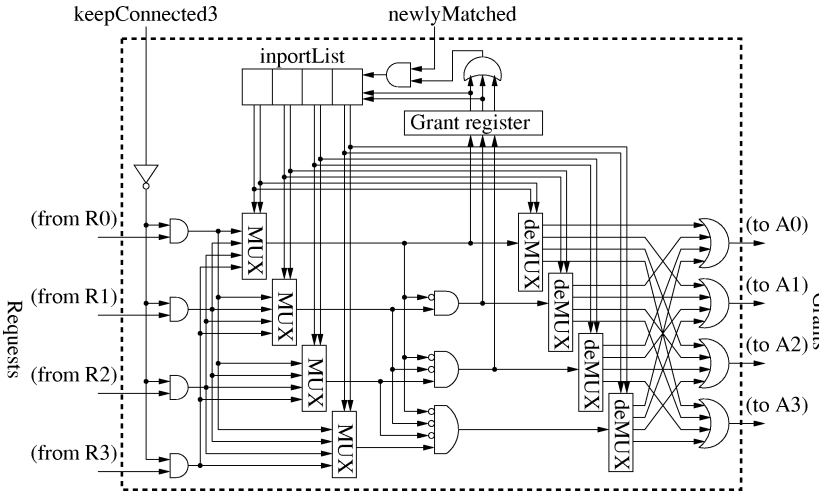

Fig. 17. iPDRR grant arbiter G_j at output_j for a 4 × 4 switch.

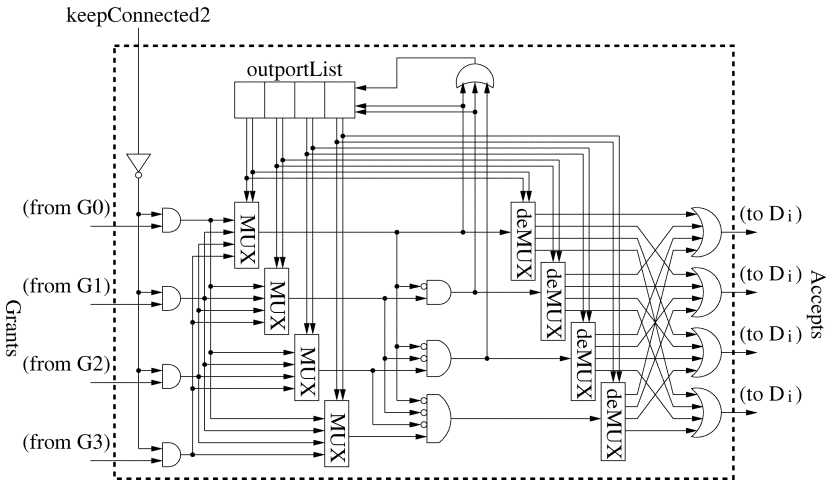

Fig. 18. iPDRR accept arbiter A_i at input_i for a 4 × 4 switch.

workload. This is because that unlike the original definition of cell-mode scheduling [17], one input–output connection can last for more than one slot even in the cell-mode iPDRR/iDRR so that the cell-mode iPDRR/iDRR in fact behaves similarly to its packet-mode counterpart.

For the nonuniform traffic, we run the simulation under the same setting as that in Section IV-D. The results are shown in Fig. 11 and Fig. 12.

Fig. 13 and Fig. 14 demonstrate iPDRR's ability of allocating bandwidth among ports in proportion to their reservations. The simulation is performed on a 4×4 switch where each input has two flows with different reservations to output 0 and output 1. When the output link speed is 1 cell/slot, all flows receive their reserved portions of the total bandwidth when the output link is overbooked. When the output link speed is reduced to about 0.5 cell/slot, the bandwidth each flow gets is still in proportion to its reservation.

VI. Hardware Implementation of iPDRR

Now let us consider the hardware implementation of iPDRR. Fig. 15 shows the block diagram of how N request modules, N grant arbiters, N accept arbiters, and N decision modules are connected to construct an iPDRR scheduler (for convenience, we only show the implementation for a 4×4 switch; it is straightforward to extend it to larger switches).

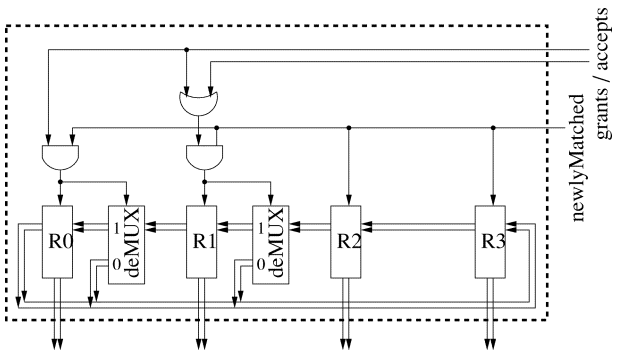

Fig. 19. iPDRR linked list for a 4 × 4 switch.

A. Request Module

The request module R_i at input_i, as illustrated in Fig. 16, is responsible for sending requests to grant arbiters G_j if input_i is not matched and VOQ_{ij} is not empty.

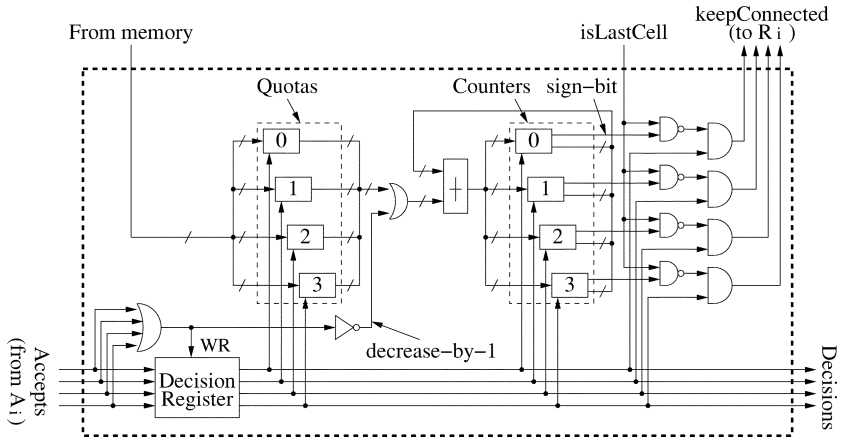

Fig. 20. iPDRR decision module D_i at input_i for a 4 × 4 switch.

Like iSLIP scheduler [22], the request–grant–accept iteration in iPDRR is also pipelined, i.e., requests and grants in the next iteration can be sent at the same time when the accept decisions in the current iteration are made. Whenever accept arbiter A_i (see Section VI-B) receives at least one grant, *anyGrant* signal is set and R_i is disabled in the next iteration.

The request module R_i also takes as inputs the input–output connection information made in the previous slot by decision module D_i (see Section VI-C). Together with the state of VOQ_{ij}, this connection information determines whether the corresponding input–output connection will be kept or not in the current slot. If so, the corresponding grant and accept arbiters are disabled in the current slot.

B. Grant/Accept Arbiter

Fig. 17 shows the diagram of a grant arbiter G_j at output_j. It maintains a linked list $\mathrm{inportList}_j$ which records the order of input ports. The job of G_j is to select, in the order of $\mathrm{inportList}_j$, the first i such that input_i sends a request, and sent a grant to input_i. It is done by N multiplexers which reorder requests according to $\mathrm{inportList}_j$ and N demultiplexers which restore the grants to their original order. The *grant register* is used to save the grant decision in the previous iteration because of the pipelined implementation. It is used to update $\mathrm{inportList}_j$ when output_j is matched.

The accept arbiter (see Fig. 18) is almost the same as the grant arbiter, except that it does not need a register to hold the accept decision to update $\mathrm{outportList}$ because an input that receives at least one grant will definitely be matched.

When input_i and output_j are matched, we need to update the two linked lists by moving j to the end of $\mathrm{outportList}_i$ and i to the end of $\mathrm{inportList}_j$. This operation is equivalent to a circular register shift.

As illustrated in Fig. 19, for a linked list of N ports, we have N registers, each $\log N$ bit wide. R_i is the ith register, initialized to i. When there is a new matching, the corresponding register, say R_i, and all registers after R_i will be enabled so that $R_i \leftarrow R_{i+1} \leftarrow R_{i+2} \leftarrow \ldots \leftarrow R_{N-2} \leftarrow R_{N-1} \leftarrow R_i$ can be done in one shot.

Note that if the last port in the list is matched, we do not have to update the list. Also, when the list needs to be updated, the last two registers will always be enabled. So there are $N-2$ grant/accept input signals instead of N.

C. Decision Module

After the accept arbiter makes a decision, the result goes to the decision module (see Fig. 20) and is stored in the decision register. The main functionality of the decision module is to perform bandwidth fair sharing among ports.

Each decision module has N registers for quotas and N registers for counters. Since updating quotas is not time critical, for each quota, we keep a copy in memory. When a flow comes and goes, the corresponding quota is updated in memory and then copied to its register. In packet-mode scheduling, *isLastCell* is set to 1 when the last cell of a packet is transferred across the switch in the current slot. In cell-mode scheduling, *isLastCell* is always set to 1.

When input_i and output_j are newly matched, $\mathrm{counter}_{ij}$ and quota_{ij} are enabled, and $\mathrm{counter}_{ij}$ is increased by quota_{ij}. After that, the connection will be kept and $\mathrm{counter}_{ij}$ will be decreased by 1 every slot until the sign bit of $\mathrm{counter}_{ij}$ and *isLastCell* are set to 1.

The *keepConnected* signals go to the request module, which further decides whether a connection needs to be kept or not in the next slot.

VII. Conclusion

In this paper, we first demonstrated that applying fair queuing only at the output link is not very effective, because the number of packets competing for the output link is limited in input-queued switches. Therefore, we proposed iDRR, a flow-based fair scheduling algorithm which can allocate the switch bandwidth in proportion to each flow's reservation. The iDRR is implemented to an iteration-based switch scheduler so that packets are properly selected from the input queues for transmission. We showed that such a scheme achieves fair scheduling while providing high throughput and low latency. Since flow-based fair scheduling schemes are difficult to implement in hardware, we also proposed a port-based fair scheduling algorithm iPDRR, and described its hardware implementation in details. We also

compared the performance of iPDRR with other schemes to demonstrate the superiority of our technique.

REFERENCES

[1] Cisco 12000 Series—Internet Routers [Online]. Available: http://www.cisco.com

[2] C. Partridge *et al*., "A 50-Gb/s IP router," *IEEE/ACM Trans. Networking*, vol. 6, no. 3, pp. 237–248, June 1998.

[3] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," *J. Internetworking Research and Experience*, vol. 1, no. 1, pp. 3–26, Sept. 1990.

[4] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Networking*, vol. 1, pp. 344–357, June 1993.

[5] L. Zhang, "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Trans. Computer Systems*, vol. 9, pp. 101–124, 1991.

[6] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," in *Proc. SIGCOMM*, Boston, MA, Aug. 1995.

[7] J. C. Bennett and H. Zhang, "$\mathrm{WF}^2\mathrm{Q}$: worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM '96*, San Francisco, CA, Mar. 1996, pp. 120–128.

[8] N. Ni and L. N. Bhuyan, "Fair scheduling and buffer management in internet routers," in *Proc. IEEE INFOCOM*, New York, June 2002, pp. 1141–1150.

[9] Y. Tamir and G. Frazier, "High performance multi-queue buffers for vlsi communication switches," in *Proc. 15th Ann. Symp. Computer Architecture*, June 1988, pp. 343–354.

[10] D. Stiliadis and A. Varma, "Providing bandwidth guarantees in an input-buffered crossbar switch," in *Proc. IEEE INFOCOM*, 1995, pp. 960–968.

[11] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High speed switch scheduling for local area networks," *ACM Trans. Comput. Syst.*, vol. 11, no. 4, pp. 319–352, Nov. 1993.

[12] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Networking*, vol. 7, pp. 188–201, Apr. 1999.

[13] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," in *IEEE INFOCOMM*, vol. 1, San Francisco, CA, Mar. 1996, pp. 296–302.

[14] A. Mekkittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *IEEE INFOCOM*, vol. 2, San Francisco, CA, Apr. 1998, pp. 792–799.

[15] M. A. Marsan, A. Bianco, E. Leonardi, and L. Milia, "RPA: a flexible scheduling algorithm for input buffered switches," *IEEE Trans. Commun.*, vol. 47, pp. 1921–1933, Dec. 1999.

[16] P. Giaccone, D. Shah, and B. Prabhakar, "An implementable parallel scheduler for input-queued switches," *IEEE Micro*, vol. 22, pp. 19–25, Jan./Feb. 1999.

[17] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Packet scheduling in input-queued cell-based switches," in *IEEE INFOCOM*, vol. 2, 2001, pp. 1085–1094.

[18] NLANR Network Traffic Packet Header Traces [Online]. Available: http://pma.nlanr.net/Traces/

[19] S. S. Kanhere, H. Sethu, and A. B. Parekh, "Fair and efficient packet scheduling using elastic round robin," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, pp. 324–336, Mar. 2002.

[20] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz, "The tiny tera: A packet switch core," *IEEE Micro*, pp. 26–33, Jan./Feb. 1997.

[21] C. S. Chang, D. S. Lee, and Y. S. Jou, "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering," *Comput. Commun.*, vol. 25, pp. 611–622, 2002.

[22] P. Gupta and N. McKeown, "Design and implementation of a fast crossbar scheduler," *IEEE Micro*, pp. 20–28, Jan./Feb. 1999.

Xiao Zhang (S'01) received the B.E. degree in computer science from Shanghai Jiao Tong University, Shanghai, P.R. China, in 1991, and the M.S. degree in computer science from University of California, Riverside, in 2001. He is currently working toward the Ph.D. degree at University of California, Riverside.

His research interests include switch scheduling, high-performance router, high availability cluster, and distributed system.

Laxmi N. Bhuyan (F'98) received the Ph.D. degree in computer engineering from Wayne State University, Detroit, MI, in 1982.

He has been Professor of Computer Science and Engineering at the University of California, Riverside, since January 2001. Prior to that, he was a Professor of Computer Science at Texas A&M University, College Station (1989–2000) and Program Director of the Computer System Architecture Program at the National Science Foundation (1998–2000). He has also worked as a Consultant to Intel and HP Labs. His research addresses multiprocessor architecture, network processors, Internet routers, web servers, parallel and distributed computing, and performance evaluation.

Dr. Bhuyan is a Fellow of the ACM and the AAAS. He has also been named as an ISI Highly Cited Researcher in Computer Science.