Adaptive Max-min Fair Scheduling in Buffered Crossbar Switches Without Speedup

Xiao Zhang, Satya R. Mohanty and Laxmi N. Bhuyan
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
Email: {xzhang, satya, bhuyan}@cs.ucr.edu

***Abstract*— A good crossbar switch scheduler should be able to achieve 100% throughput and maintain fairness among competing flows. A pure input-queued (IQ) non-buffered switch requires an impractically complex scheduler to achieve this goal. Common solutions are to use crossbar speedup and/or buffered crossbar.**

In this paper, we explore this issue in a buffered crossbar without speedup. We first discuss the conflict between fairness and throughput and the fairness criteria in crossbar switch scheduling, and justify that a desirable scheduler should sustain full bandwidth for admissible traffic and ensure max-min fairness for non-admissible traffic. Then we describe an *adaptive max-min fair scheduling* (AMFS) algorithm and show by analysis and simulation that it can provide both 100% throughput and max-min fairness. Finally we briefly discuss the hardware implementation of the AMFS algorithm.

***Index Terms*— switch scheduling, buffered crossbar, combined input crosspoint queued (CICQ) switch, quality of service (QoS), max-min fairness.**

I. Introduction

Input-queued (IQ) crossbar switch scheduling has been a topic for decades. However, the challenges still remain partly because the network bandwidth is increasing rapidly, and partly because of the intricate difficulty of the crossbar scheduling.

The problem of IQ crossbar scheduling can be formalized as the classical graph theory problem of maximum weight matching on the bipartite graph where nodes represent input and output ports, and edges represent packets to be switched. The maximum weight matching (MWM) algorithm[1] has been proved to achieve 100% throughput [1] but is too complex for fast hardware implementation.

For an algorithm to be practical, it must be fast. For example, with 10-Gbps (approximately OC-192) line card speed and 64-byte packet size, a scheduling decision is preferred to be made within 51.2 ns. Heuristic algorithms, such as iSLIP [2], iFS [3], iDRR [4], meet the time-constraint, but fail to provide 100% throughput for admissible[2] non-uniform traffic. One approach is to apply moderate crossbar speedup (the ratio of the crossbar speed and line card speed). An exciting result is that any maximal algorithms with a speedup of 2 can support 100% throughput [5]. However, the downside of this approach is that doubling crossbar speed requires memory speed to be doubled and scheduling time to be halved.

Recently, with the breakthrough of very large scale integration (VLSI) and application specific integrated circuit (ASIC) technology, a large amount of buffer can be easily integrated into a single chip. This makes buffered crossbar (a small buffer resides at each crosspoint) a very promising solution. Although the number of buffers is N^2 (where N is the crossbar size), nowadays, it is not the memory but the number of pins that dominates the chip area.

A big advantage of a buffered crossbar is the simplification of the scheduling algorithm. The crosspoint buffers separate the input contentions from the output contentions so that each input and output arbiter can work independently.

Early studies demonstrated by simulation that a buffered crossbar switch provides better throughput than an non-buffered crossbar switch with much simpler schedulers, such as oldest cell first (OCF)-OCF [6] and round-robin (RR)-RR [7] Later, longest queue first (LQF)-RR [8] has been proved to achieve 100% throughput for uniform admissible traffic. More recently, Shortest crosspoint buffer first (SCBF) [9] has also been proved to support 100% throughput for any admissible traffic. Unfortunately, these algorithms fail to provide fairness. On the other hand, by applying a *packet fair queuing* (PFQ) algorithm [10]–[13] at each input and output, PFQ-PFQ has been shown to provide fairness among competing flows [14]; but as we will show later, it fails to sustain full bandwidth under admissible non-uniform traffic.

To provide both throughput and quality of service (QoS), researches again resort to speedup. Magill *et al.* show how to emulate an OQ switch with speedup of 2 [15]. And a more recent work by Chuang *et al.* further describes a set of scheduling algorithms to provide throughput, rate and delay guarantees with speedup of 2 or 3 [16].

One open question is: can we achieve both throughput and fairness in a buffered crossbar without speedup? Our answer is both yes and no. It is no because it is impossible to achieve both 100% throughput and strict fairness at the same time (even in the case of admissible traffic!) due to the coexistence of input and output contentions. On the other hand, the answer is yes because if we relax the strict fairness criterion to a dynamic max-min fairness, fairness implies 100% throughput for admissible traffic. Suprisingly enough, in a buffered crossbar switch, this fairness can be achieved with a

[1]The MWM algorithm assigns each VOQ_{ij} a weight w_{ij}, and finds a matching M that maximizes $\sum_{(i,j)\in M} w_{ij}$. The weight can be queue length, waiting time or others.

[2]Admissibility will be defined subsequently in section IV.

very simple (and almost obvious) modification to an existing algorithm: a PFQ-PFQ scheme with dynamic weights based on both queue lengths and assigned weights. We name this algorithm *adaptive max-min fair scheduling* (AMFS), provide a detailed description, present analysis and simulation results, and finally discuss its hardware implementation issues.

The rest of the paper is organized as follows. In section II, we briefly overview the switch model used in this paper. In section III, we discuss the relationship between fairness and throughput in the context of crossbar scheduling without speedup. In section IV, V, VI and VII, we describe our main algorithms, provide the max-min fairness definition, and present the throughput and fairness analysis. In section VIII, we show simulation results to verify our scheme and compare with existing schemes. In section IX, we briefly discuss the hardware implementation of the AMFS algorithm. Finally in section X we present our conclusions.

II. Background: switch model

Fig. 1 shows a high-level diagram of an input-queued non-buffered crossbar switch. The crossbar operates on fixed-size packets (called *cells*) at the same speed as line cards. Time is divided into *time slots*, and it takes one slot to transfer one cell. Variable-size packets are segmented at inputs and reassembled at outputs. To avoid the head-of-line (HOL) blocking [17], virtual output queuing (VOQ) [18] is used, where a logical separate FIFO queue is maintained for each input-output pair. Because of both input and output contentions, a crossbar scheduler is necessary to decide which cells are transferred across the crossbar in the next slot.

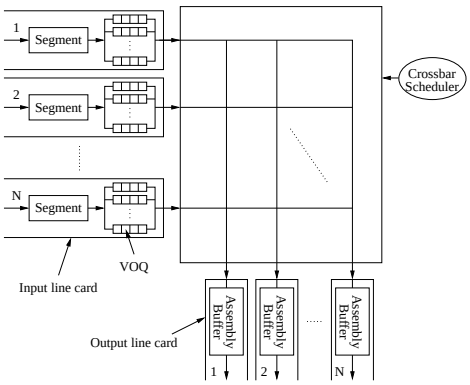

Fig. 1. An $N \times N$ non-buffered crossbar switch.

In a buffered crossbar switch, a small buffer, called crosspoint buffer (CB), is put at each crosspoint, as shown in Fig. 2. From the queuing point of view, this switch architecture is called *combined input crosspoint queued* (CICQ) switch. Note that if the crosspoint buffer is infinite or very large, CICQ is equivalent to output queuing and input arbiters are not necessary since packets can be directly stored at the crosspoint buffer upon their arrival. To make a single chip implementation feasible, the crosspoint buffer has to be limited, and therefore imposes a challenge to scheduling. A great benefit of a buffered crossbar is that the scheduling becomes much simpler. Instead of considering inputs and outputs at the same time, a buffered crossbar allows input and output arbiters to work independently. In this paper, we address the problem of how to achieve throughput and fairness in a buffered crossbar switch without speedup. In the next section, we first examine the problem and present the motivation behind this work.

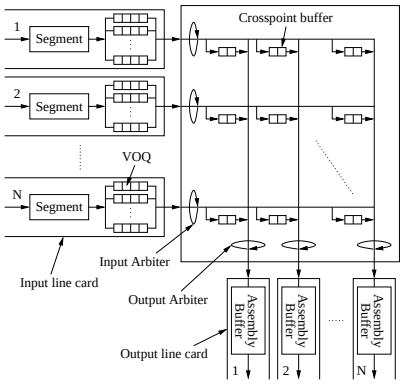

Fig. 2. An $N \times N$ buffered crossbar switch.

III. Motivation: Throughput vs. Fairness

With appropriate crossbar speedup and scheduler, an IQ/CICQ switch can emulate an OQ switch [15], [19], which means that throughput and fairness can be satisfied at the same time. However, in the case of no speedup, this is not the case. Note that our discussion in this section applies to both buffered and non-buffered crossbar.

Fairness implies equal allocation of resources, but there is little agreement among researchers as to what needs to be equalized. The best-effort traffic mainly refers to those that are not delay sensitive. Therefore, in this paper, by fairness, we mean fair sharing bandwidth among competing flows.

In general, the goal of achieving fairness conflicts with the goal of maximizing throughput. Consider three backlogged flows f_{11}, f_{12} and f_{22} going through a 2×2 crossbar, where f_{ij} is the flow from input i to output j. If we want to maximize the overall crossbar throughput, the only choice is to schedule f_{11} and f_{22}, as shown in Fig. 3, so that the throughput is 2. Clearly such scheduling starves f_{12}. If we want to fairly treat each flow, we have to schedule each with the per-flow rate of 0.5, where the overall throughput is only 1.5. Even though the output 1 is idle when f_{12} is scheduled, the second scheduling strategy avoids starvation and enforces fairness. Fig. 4 shows the throughput of three flows as a function of workload using OCF[3] [20]. When workload is above 0.5, f_{12} receives much less than other two flows. Although 100% throughput is achieved, this case shows that it is necessary to provide overload protection to ensure fairness. Otherwise, a malicious user can easily steal bandwidth by flooding the network.

[3]Oldest Cell First (OCF) is a MWM algorithm with weight defined as waiting time of the head-of-line (HOL) cell. Note it is different from that in the OCF-OCF algorithm [6].

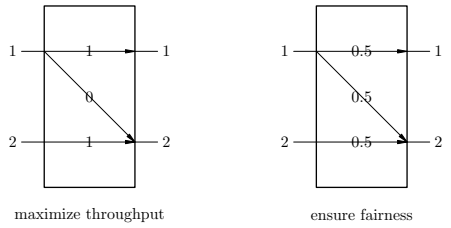

Fig. 3. Illustration of conflict between throughput and fairness.

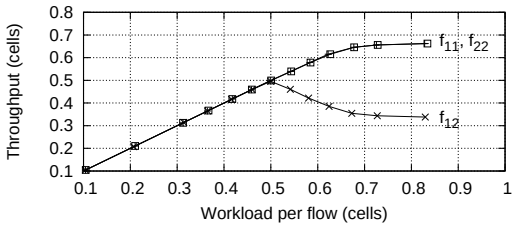

Fig. 4. OCF fails to ensure fairness under non-admissible traffic.

Notice that in many switch scheduling studies, it is assumed that at most 1 cell arrives at each input in one slot. Under this assumption, an input can never be over-subscribed. For discussion of non-admissible traffic, we remove this assumption, allowing more than 1 cells arrive at each input in one slot. Thus the aggregate arrival rate at each input can be greater than 1, as shown in Fig. 4. Non-admissible traffic includes input over-subscription as well as output over-subscription. Although allowing input over-subscription sounds unnecessary in a crossbar switch because the switch can only transfer 1 cell per input per slot, it is still practical. For example, an input can be an aggregate link of many links or queues, and the aggregate arrival rate can be much higher than 1 cell per slot. If only 1 cell is allowed to arrive at one slot, we must put another scheduler at the aggregate link to schedule cells from the aggregate link to VOQs. This scheduler is redundant because the switch scheduler can do the same thing.

In the case of admissible traffic, strict fairness can still lead to bandwidth under-utilization. To illustrate this problem, let's look at the example shown in Fig. 5. We have three flows f_{11}, f_{21} and f_{22}, and c_{ij}^k denotes the k^{th} cell of f_{ij}. Cells of f_{11} arrive at slot $3n-2$; cells of f_{21} arrive at slot $3n-2$ and $3n-1$; and cells of f_{22} arrive at slot $3n$, where $n = 1, 2, 3, \ldots$. Clearly, this is an admissible traffic. Now suppose each flow has the same weight and we use an absolutely fair fluid model scheduler. In slot 1 and 2, c_{11}^1 and c_{21}^1 are transferred, and each cell gets half bandwidth and takes 2 slots. In slot 3, c_{21}^2 and c_{22}^1 are transferred, and again each gets half bandwidth. Note that in slot 3, half of the output 1 bandwidth is wasted as indicated as the gray area in the graph. Therefore, with this arrival pattern, f_{21} only receives $\frac{1}{2}$ of the bandwidth, much less than its workload of $\frac{2}{3}$, and its queue will gradually become infinite. There is a waste of $\frac{1}{6}$ of the bandwidth at output 1, but we cannot use them because of the fairness concern.

This example again shows that achieving both absolute

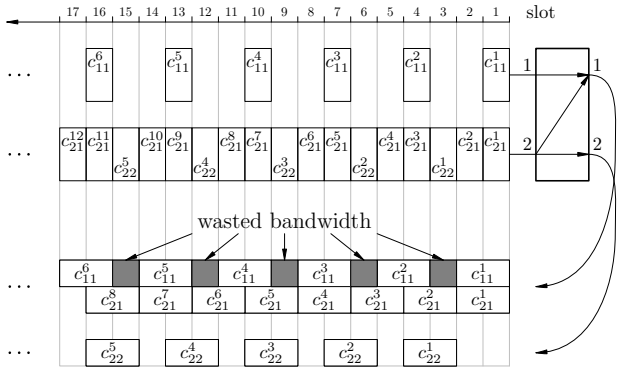

Fig. 5. Strict fairness leads to under-utilization of bandwidth for admissible traffic.

fairness and 100% throughput at the same time is impossible. Therefore, some trade-off must be made. However, unlike the case in Fig. 3, we prefer throughput than fairness. In this particular case, we can give priority to flow f_{21} by postponing each cell of f_{11} by 2 slots. Although instantaneous fairness is not maintained, this trade-off is justifiable. First best-effort traffic is not delay sensitive, and throughput is more important than delay. Because traffic is admissible, giving preference to heavy-backlogged queues will not affect the throughput of light-backlogged queues. In a relatively long interval, fairness is still maintained. Second, although cells of light-backlogged flow incur longer delay, the increased delay only affects the initial delay observed by the end-user.

Note that fairness does not necessarily mean equal distribution of resources. In many cases, it is justifiable to give more bandwidth to some flows than others. How to assign weights depends on applications. From now on, we assume that each VOQ_{ij} is assigned a weight w_{ij}.

So far, we haven't formulated the fairness criterion when a crossbar is overloaded. At first glance, it is appealing to use the fairness definition of the well known GPS system [10], i.e., for any competing flow i and j that are continuously backlogged in the interval $[\tau, t)$, $\frac{W_i[\tau,t)}{W_j[\tau,t)} = \frac{\phi_i}{\phi_j}$, where $W_i[\tau, t)$ is the amount of flow i traffic served in an interval $[\tau, t)$ and ϕ_i is the weight of flow i. However, when in the case of a crossbar, this definition in general does not hold without unnecessarily wasting bandwidth. In fact, in terms of bandwidth distribution, it is more appropriate to represent crossbar as a network as shown in Fig. 6.

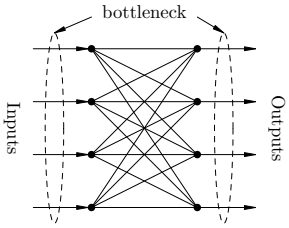

Fig. 6. Crossbar network.

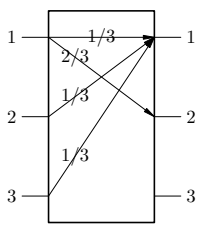

Fig. 7. Max-min bandwidth distribution in a crossbar.

Now it is clear that there are $2N$ bottlenecks. Different flows might have different bottlenecks. In this scenario, a natural fair allocation scheme called *max-min fairness* [21] can be used. The goal of max-min fairness is to achieve fairness among competing flows while not unnecessarily wasting bandwidth, i.e., maximize the minimum service rate of each flow. Fig. 7 illustrates the max-min bandwidth allocation. We have four equal-weight flows f_{11}, f_{21}, f_{31} and f_{12}. Although each flow has the same weight, f_{12} can have $\frac{2}{3}$ of the total bandwidth without affecting other flows' throughput. To summarize, a desirable scheduler should fulfill the following:

1) sustain 100% throughput for admissible traffic, and
2) ensure max-min fairness for non-admissible traffic.

In the next 4 sections, we show how to achieve the above objective in a buffered crossbar employing a very simple scheme that requires no speedup. First in section IV, based on a packet fair queuing (PFQ) algorithm, we present a queue length driven packet fair queuing (QLD-PFQ) algorithm and prove that it provides 100% throughput. Then in section V, we formally define a dynamic max-min fairness criterion for crossbar switches. In section VI, we describe an adaptive max-min fair scheduling (AMFS) algorithm and show its max-min fairness property. Finally in section VII, we apply AMFS to the case of finite buffers.

IV. Queue Length Driven Packet Fair Queuing

In [14], Stephens and Zhang study a distributed packet fair queuing (D-PFQ) system, where each input and output apply PFQ independently, hence we refer to this scheme as PFQ-PFQ in this paper. It is shown that PFQ-PFQ provides fairness among competing flows. In [22], PFQ-PFQ is shown to automatically converge to the max-min fair rate allocation under a fully overloaded situation. [23] further shows that by viewing the scheduling as a non-cooperative strategic game, the max-min fair allocation is equivalent to the Nash equilibrium, thus providing a solid foundation for the max-min fairness criterion.

However, PFQ-PFQ cannot sustain full bandwidth when traffic is admissible. To see why this is the case, assume all flows have the same weight, then PFQ-PFQ becomes equivalent to RR-RR which is already demonstrated in [7], [8] to fail to provide 100% throughput under admissible non-uniform traffic. In fact, we've already shown in section III that an algorithm focusing on fairness alone fails to sustain 100% throughput because of the conflict between throughput and fairness.

The underlying reason that PFQ-PFQ fails to sustain full bandwidth under admissible traffic is that it doesn't take queue status into consideration. This motivates us to use queue length as weight in the scheduling decision. Formally, let $\phi_{ij}(t)$ be the weight of queue ij (from input i to j at time t, define

$$\phi_{ij}(t) = x_{ij}(t) \tag{1}$$

where $x_{ij}(t)$ be length of queue ij (combined queue length of VOQ_{ij} and CB_{ij}) at time t. We call PFQ-PFQ with the above weight definition *queue-length-driven packet fair queuing* (QLD-PFQ). Now we show that QLD-PFQ achieves 100% throughput. We will adopt the notation and definitions introduced in [24]. For a $N \times N$ switch, define

- X_n: the vector of queue lengths at time n, i.e., $X_n = (x_n^{11}, \ldots, x_n^{ij}, \ldots, x_n^{NN})$, where x_n^{ij} is the $((i-1)*N+j)^{th}$ element, denoting the length of queue ij at time n. Assume $x_0^{ij} = 0$.
- A_n: the vector of numbers of arrivals at time n, i.e., $A_n = (a_n^{11}, \ldots, a_n^{ij}, \ldots, a_n^{NN})$, where a_n^{ij} is the number of arrivals at queue ij in the time interval $(n, n+1]$. Assume that the arrival process A_n is independent and identically distributed.
- D_n: the vector of numbers of departures at time n, i.e., $D_n = (d_n^{11}, \ldots, d_n^{ij}, \ldots, d_n^{NN})$, where d_n^{ij} is the number of departures from queue ij in time interval $(n, n+1]$. Hence, the evolution of the system of queues can be described as:

$$X_{n+1} = X_n + A_n - D_n \tag{2}$$

- Λ: the vector of average arrival rates, i.e., $\Lambda = (\lambda_{11}, \ldots, \lambda_{ij}, \ldots, \lambda_{NN})$, where λ_{ij} is the arrival rate at queue ij. Clearly, $E[A_n] = \Lambda$. A traffic is said to be *admissible* if no input or output is oversubscribed, i.e.,

$$\sum_{k=1}^{N} \lambda_{ik} \leq 1 \quad \text{and} \quad \sum_{k=1}^{N} \lambda_{kj} \leq 1 \tag{3}$$

- M: the vector of average departure (or service) rates, i.e., $M = (\mu_{11}, \ldots, \mu_{ij}, \ldots, \mu_{NN})$, where μ_{ij} is the departure rate from queue ij. Clearly, $E[D_n] = M$.
- $||X||$: Euclidean norm of vector $X = (x_1, x_2, \ldots, x_K)$. i.e., $||X|| = \sqrt{\sum_{i=1}^{K} x_i^2}$.

Definition 1: (throughput) A system of queues is strongly stable (implies 100% throughput) if for any admissible traffic, $\lim_{n\to\infty} \sup E[||X_n||] < \infty$.

This definition basically means that in a strongly stable system, the average queue length and hence the average queue delay is bounded. To prove that QLD-PFQ leads to a stable system, we first introduce the following lemma.

Lemma 1: A switch with a scheduling algorithm such that $E[D_n] = (1+\alpha)\tilde{X}_n + D'_n$ is strongly stable, where $\alpha \in \mathbb{R}^+$ ($\mathbb{R}^+$ is the set of non-negative real number), $D'_n \in \mathbb{R}^{+N^2}$ and is a function of X_n, and $\tilde{X}_n$ is the normalized vector of X_n, i.e., $\tilde{X}_n = \frac{X}{\max_{ij}(\sum_k x_n^{ik}, \sum_k x_n^{kj})}$.

Proof: refer to theorem 6, 7 and 8 in [24]. ■

Lemma 1 states that if the departure rate is parallel to the queue length X, and longer than $\tilde{X}$, the system is stable. We show that in a buffered crossbar employing QLD-PFQ, the departure rate indeed satisfies Lemma 1.

Lemma 2: In a buffered crossbar switch operating under the QLD-PFQ algorithm, $E[D_n] = \tilde{X} + D'_n$ for any admissible traffic.

Proof: Given a buffered crossbar, consider its corresponding fluid model using the GPS scheduler [10]. Let $z_{ij}(t)$ be the queue level of *VOQ*$_{ij}$ at time t, $b_{ij}(t)$ be the queue level of *CB*$_{ij}$ at time t, and $x_{ij}(t) = z_{ij}(t) + b_{ij}(t)$.

According to equation (1), $\phi_{ij}(t) = x_{ij}(t)$. Therefore, with PFQ, at each input, the service rate $\mu_{ij}^{in}(t)$ for queue ij at time t is

$$\mu_{ij}^{in}(t) = \frac{x_{ij}(t)}{\sum_k x_{ik}(t)} \tag{4}$$

Similarly, at each output, the service rate $\mu_{ij}^{out}(t)$ for queue ij at time t is

$$\mu_{ij}^{out}(t) = \frac{x_{ij}(t)}{\sum_k x_{kj}(t)} \tag{5}$$

Because inputs and outputs are coupled by crosspoint buffers,

$$\begin{aligned} \mu_{ij}(t) &= \min(\mu_{ij}^{in}(t), \mu_{ij}^{out}(t)) + d_{ij}^m(t) \\ &= \min(\frac{x_{ij}(t)}{\sum_k x_{ik}(t)}, \frac{x_{ij}(t)}{\sum_k x_{kj}(t)}) + d_{ij}^m(t) \end{aligned} \tag{6}$$

where $d_{ij}^m(t) \in \mathbb{R}^+$ indicates the extra service due to the fact that GPS automatically converges to max-min rate, and convergence takes finite time depending on the CB size [22]. Let $d_{ij}^e(t) = \min(\frac{x_{ij}(t)}{\sum_k x_{ik}(t)}, \frac{x_{ij}(t)}{\sum_k x_{kj}(t)}) - \frac{x_{ij}(t)}{\max_{ij}(\sum_k x_{ik}(t), \sum_k x_{kj}(t))}$. Obviously, $d_{ij}^e(t) \geq 0$.

$$\begin{aligned} \mu_{ij}(t) &= \frac{x_{ij}(t)}{\max_{ij}(\sum_k x_{ik}(t), \sum_k x_{kj}(t))} + d_{ij}^e(t) + d_{ij}^m(t) \\ &= \tilde{x}_{ij}(t) + d_{ij}^e(t) + d_{ij}^m(t) \end{aligned} \tag{7}$$

Finally, let $d'_{ij}(t) = d_{ij}^e(t) + d_{ij}^m(t)$, we have

$$E[D_n] = \tilde{X}_n + D'_n \tag{8}$$

■

Theorem 1: A buffered crossbar switch operating under the QLD-PFQ algorithm is strongly stable.

Proof: straightforward from Lemma 1 and 2. ■

V. MAX-MIN FAIRNESS CRITERION

Unfortunately, QLD-PFQ cannot provide fairness when the switch is over-loaded because it doesn't take into consideration the pre-assigned weight of each queue. To discuss fairness, we need to formally define fairness. First, we introduce the max-min fairness criteria in a $N \times N$ crossbar switch based on the general definition in [21]. Let

- f_{ij} be the flow from *VOQ*$_{ij}$. We say f_{ij} is *idle* if $\lambda_{ij} = 0$, and f_{ij} is *active* if $\lambda_{ij} > 0$. Note *VOQ*$_{ij}$ can be temporarily empty even if f_{ij} is active.
- $W = [w_{ij}]$ be the matrix of assigned weights of N^2 flows. Assume $w_{ij} = 0$ if f_{ij} is idle, and $w_{ij} > 0$ if f_{ij} is active.
- $\check{C} = (\check{c}_i)$ be the vector of input capacities.
- $\hat{C} = (\hat{c}_j)$ be the vector of output capacities. For a switch without speedup, $\check{c}_i = \hat{c}_j = 1, \forall i, j$.
- $R = [r_{ij}]$ be the matrix of rate allocation for each flow.

We call R feasible if

$$\sum_{k=1}^{N} r_{ik} \leq \check{c}_i \quad \text{and} \quad \sum_{k=1}^{N} r_{kj} \leq \hat{c}_j \tag{9}$$

We also call a feasible allocation R weighted max-min fair, when it is impossible to increase r_{ij} without losing feasibility or reducing r_{pq} satisfying $\frac{r_{pq}}{w_{pq}} \leq \frac{r_{ij}}{w_{ij}}$.

Given W, $\check{C}$ and $\hat{C}$, it is easy to find the max-min rate matrix R using the water-filling approach [21]. However, the rates calculated in this way only represent the maximum throughput each flow gets when all active flows are continuously backlogged. If a flow only uses some portion of its max-min rate, the unused portion should be used by other flows. Naturally we want this unused bandwidth to be distributed in the max-min fair manner.

To take this situation into consideration, we further introduce *dynamic max-min rate matrix* $R^{w\lambda} = [r_{ij}^{w\lambda}]$, based on both W and Λ, whereas the matrix, based only on W, is referred to as *static max-min rate matrix* $R^w = [r_{ij}^w]$. The fairness the two matrices represent is called *dynamic max-min fairness* and *static max-min fairness* respectively. Given W, $\check{C}$, $\hat{C}$ and Λ, the following procedure gives $R^{w\lambda}$.

1) $R^{w\lambda} \leftarrow \varnothing$
2) calculate R^w (note this R^w is a local variable)
3) for each $\lambda_{ij} \leq r_{ij}^w$ do
   $r_{ij}^{w\lambda} \leftarrow \lambda_{ij}$
   $\check{c}_i \leftarrow \check{c}_i - \lambda_{ij}$
   $\hat{c}_j \leftarrow \hat{c}_j - \lambda_{ij}$
   $w_{ij} \leftarrow 0$
4) repeat 2) and 3) until $\lambda_{ij} > r_{ij}^w$ or $w_{ij} = 0, \forall i, j$
5) for each $r_{ij}^w > 0$ and $r_{ij}^{w\lambda} = 0$ do
   $r_{ij}^{w\lambda} \leftarrow r_{ij}^w$

Clearly every non-zero entry in a $R^{w\lambda}$ falls into the following two cases:

- $\lambda_{ij} > r_{ij}^{w\lambda}$: in this case, we call f_{ij} a *non-admissible* flow.
- $\lambda_{ij} = r_{ij}^{w\lambda}$: in this case, we call f_{ij} an *admissible* flow. Furthermore, if $r_{ij}^{w\lambda} \leq r_{ij}^w$, we call f_{ij} an *absolutely admissible* flow; whereas if $r_{ij}^{w\lambda} > r_{ij}^w$, f_{ij} is called a *relatively admissible* flow. because in the latter case, a flow is good because another admissible flow doesn't use up its max-min bandwidth.

Note, if we remove an absolutely admissible flow from the crossbar network, another relatively admissible flow may become an absolutely admissible flow. To convey this observation formally, let $\Psi = (W, \Lambda, \check{C}, \hat{C})$, and recursively define $\Psi^k = (W^k, \Lambda^k, \check{C}^k, \hat{C}^k)$ as follows:

$$\begin{array}{llll} w_{pq}^k = w_{pq}^{k-1}, & & w_{ij}^k = 0, \\ \lambda_{pq}^k = \lambda_{pq}^{k-1}, & & \lambda_{ij}^k = 0, \\ \check{c}_i^k = \check{c}_i^{k-1} - \lambda_{ij}, & & \hat{c}_j^k = \hat{c}_j^{k-1} - \lambda_{ij}, \end{array}$$

where $k \geq 1$, $p \neq i$, $q \neq j$ and f_{ij} is an absolutely admissible flow with respect to Ψ^{k-1}. It is easy to get the following facts:

1) $r_{ij}^{w^k\lambda^k} = r_{ij}^{w^{k-1}\lambda^{k-1}}$, $\forall r_{ij}^{w^k\lambda^k} > 0$ (i.e., the dynamic max-min rate in Ψ^k is the same as that in Ψ^{k-1}).
2) $r_{ij}^{w^k} \geq r_{ij}^{w^{k-1}}$, $\forall r_{ij}^{w^k} > 0$ (i.e., the static max-min rate in Ψ^k is at least the same as that in Ψ^{k-1}).
3) $\{f_{ij}|\lambda_{ij} = r_{ij}^{w^k\lambda^k}\} \neq \varnothing \Rightarrow \exists f_{ij}$ such that $\lambda_{ij} \leq r_{ij}^{w^k}$ (i.e., if there are admissible flows in Ψ^k, there must exist an absolutely admissible flow with respect to Ψ^k).

Definition 2: (fairness) Given weight W and arrival rate Λ, a crossbar switch under a scheduling algorithm is fair if $\mu_{ij} = r_{ij}^{w\lambda}$, where μ_{ij} is the departure (or service) rate for f_{ij}.

This definition simply states that for admissible flows, the service rate is equal to the workload; and for non-admissible flows, the service rate is its dynamic max-min rate. Note that by using dynamic max-min fairness criterion, this fairness definition implies 100% throughput for admissible traffic. In fact, if $\lambda_{ij} = r_{ij}^{w\lambda}, \forall$ active $f_{ij} \Rightarrow \sum_k \lambda_{ik} = \sum_k r_{ik}^{w\lambda} \leq \check{c}_i = 1$ and $\sum_k \lambda_{kj} = \sum_k r_{kj}^{w\lambda} \leq \hat{c}_j = 1$ (i.e., if all active flows are admissible, the traffic is admissible).

VI. The adaptive max-min fair scheduling (AMFS) algorithm

Taking the above max-min fairness definition into consideration, we now extend the weight definition in equation (1) as follows:

$$\phi_{ij} = \begin{cases} \frac{x_{ij}}{N} & 0 \leq x_{ij} \leq N \\ 1 + (x_{ij} - N)\frac{w_{ij}-1}{N} & N < x_{ij} < 2N \\ w_{ij} & 2N \leq x_{ij} \leq \infty \end{cases} \quad (10)$$

where N is a large number such that for every $\epsilon > 0$, $\lim_{n\to\infty} Pr\{||X_n|| > N\} < \epsilon$ when the switch under the QLD-PFQ algorithm is loaded with any admissible traffic. This N exists because the switch is strongly stable under QLD-PFQ. In addition, to make sure that ϕ_{ij} is non-decreasing as a function of queue length, we also scale all w_{ij} in proportion so that $w_{ij} > 1, \forall i, j$. We call PFQ-PFQ using this weight definition *adaptive max-min fair scheduling* (AMFS).

When a flow's queue length is below N, the flow is regarded as admissible, and ϕ_{ij} is in proportion to its queue length. Note, it is possible that a flow is admissible when its queue length grows beyond N. Thanks to the strong stability of QLD-PFQ with infinite queue and the asymptotic construction of N, the probability of queue length greater than N can be asymptotically small.

When a flow's queue length is between N and $2N$, the flow is at the boundary between admissible and non-admissible, ϕ_{ij} is increased as a combination of queue length and assigned weight. The main purpose of this region is to provide a transition from admissible to non-admissible traffic so that ϕ_{ij} can smoothly change from queue length to assigned weight.

When a flow's queue length reaches $2N$, the flow is regarded as non-admissible, and $\phi_{ij} = w_{ij}$ which is the flow's assigned weight. The PFQ-PFQ algorithm will ensures max-min fairness among competing flows as discussed before. Later we will see that $2N$ also serves to protect well-behaved flows from overflow.

Now we show informally that AMFS is fair. There are 3 cases:

1) $\forall$ active f_{ij}, $\lambda_{ij} = r_{ij}^{w\lambda}$ (i.e., all flows are admissible): because $\sum_k \lambda_{ik} = \sum_k r_{ik}^{w\lambda} \leq \check{c}_i = 1$ and $\sum_k \lambda_{kj} = \sum_k r_{kj}^{w\lambda} \leq \hat{c}_j = 1$, this is the admissible traffic which approximately corresponds to the first case of $0 \leq x_{ij} \leq N$ in equation (10), where the weight of each queue is proportional to its queue length. According to the asymptotic construction of N, this approximation is accurate with probability $1 - \epsilon$, for every $\epsilon > 0$. Therefore, from theorem 1, asymptotically the switch is strongly stable, i.e., $\mu_{ij} = \lambda_{ij} = r_{ij}^{w\lambda}$.

2) $\forall$ active f_{ij}, $\lambda_{ij} > r_{ij}^{w\lambda}$ (i.e., all flows are non-admissible): obviously, in this case, $r_{ij}^{w\lambda} = r_{ij}^{w}$. This is the fully overloaded case (all queues are continuously backlogged), and corresponds to the third case of $2N \leq x_{ij} \leq \infty$, where the weight of each queue is its pre-assigned weight. [22] has proved that bandwidth will be distributed in the max-min fair fashion, i.e., $\mu_{ij} = r_{ij}^{w} = r_{ij}^{w\lambda}$.

3) $\exists$ active f_{ij}, $\lambda_{ij} = r_{ij}^{w\lambda}$ and ($\sum_k \lambda_{ik} > 1$ and/or $\sum_k \lambda_{kj} > 1$) (i.e., some flows are admissible, but their ports are overloaded): in this case, there must exist an absolutely admissible flow, say f_{pq} such that $\lambda_{pq} \leq r_{pq}^{w}$ (from fact 3). We claim that $x_{pq} \leq 2N$ because if $x_{pq} > 2N$, f_{pq} will receive its max-min service rate r_{pq}^{w}, but $\lambda_{pq} \leq r_{pq}^{w}$. Therefore $\mu_{pq} = \lambda_{pq} = r_{pq}^{w\lambda}$.

Now let $\Psi^0 = (W, \Lambda, \check{C}, \hat{C})$, and suppose Ψ^{k-1} has an absolutely admissible flow, we can remove it to obtain Ψ^k. Then in Ψ^k, if there are still admissible flows, there must exist a flow f_{pq} which is absolutely admissible with respect to Ψ^k (from fact 3). Note it maybe a relatively admissible flow with respect to Ψ^0. Again, $x_{pq} \leq 2N$, and $\mu_{pq} = \lambda_{pq} = r_{pq}^{w^k\lambda^k} = r_{pq}^{w\lambda}$. So, by induction, for all admissible flows f_{pq}, we have $\mu_{pq} = \lambda_{pq} = r_{pq}^{w\lambda}$.

If all flows in Ψ^k are non-admissible ($\lambda_{pq}^k > r_{pq}^{w^k\lambda^k}$), we have the same situation as case 2. Therefore $\mu_{pq} = r_{pq}^{w^k\lambda^k} = r_{pq}^{w\lambda}$.

VII. The AMFS algorithm with finite queue

In previous discussions, queues are assumed infinite which is impractical. In this section, we apply AMFS to the case of finite buffers. We normalize queue lengths, set two thresholds α and β ($0 < \alpha < \beta < 1$), and define ϕ_{ij} as follows:

$$\phi_{ij} = \begin{cases} l_{ij} & 0 \leq l_{ij} \leq \alpha \\ \alpha + (l_{ij} - \alpha)(\frac{w_{ij}-\alpha}{\beta-\alpha}) & \alpha < l_{ij} < \beta \\ w_{ij} & \beta \leq l_{ij} \leq 1 \end{cases} \quad (11)$$

Clearly, α and β correspond to N and $2N$ in equation (10). Note that α should be set large enough to accommodate reasonable traffic burst. This implies that the queue capacity is large enough. This assumption is valid for today's high-performance routers where each line card can easily contains buffers of capacities in hundreds of megabytes.

Note, we also replace x_{ij} (combined length of VOQ_{ij} and CB_{ij}) with l_{ij} (length of VOQ_{ij}). Since CBs are very small compared to VOQs, this modification won't affect performance. In fact, it makes input arbiters work-conserving.

A. Description of AMFS

For the convenience of the discussion of hardware implementation in section IX, we present the algorithm in detail followed by an explanation.

In an $N \times N$ buffered crossbar switch, for each input i ($1 \leq i \leq N$), we keep a set of following counters:

- *vtime*$_i$: system virtual time for input i.
- w_{ij}: assigned weight for *VOQ*$_{ij}$, ($1 \leq j \leq N$).
- l_{ij}: current queue length for *VOQ*$_{ij}$.
- ϕ_{ij}: instantaneous weight for *VOQ*$_{ij}$.
- *stime*$_{ij}$: virtual start time for *VOQ*$_{ij}$.
- *ftime*$_{ij}$: virtual finish time for *VOQ*$_{ij}$.
- *valid*$_{ij}$: boolean values to indicate if *stime*$_{ij}$ and *ftime*$_{ij}$ have been updated. Initialized to **false**.
- *req*$_{ij}$: boolean values to indicate if *VOQ*$_{ij}$ is not empty and *CB*$_{ij}$ is not full.

Then in each slot, do

Step 1: calculate each VOQ's virtual start and finish time.

1) **for** each j **do**
2) $\phi_{ij} \leftarrow 0$
3) *req*$_{ij} \leftarrow$ **false**
4) **if** *VOQ*$_{ij}$ is not empty **and** *CB*$_{ij}$ is not full, **then**
5) *req*$_{ij} \leftarrow$ **true**
6) $$\phi_{ij} = \begin{cases} l_{ij} & 0 \leq l_{ij} \leq \alpha \\ \alpha + (l_{ij} - \alpha)(\frac{w_{ij}-\alpha}{\beta-\alpha}) & \alpha < l_{ij} < \beta \\ w_{ij} & \beta \leq l_{ij} \leq 1 \end{cases}$$
7) **if** *valid*$_{ij}$ = **false then**
8) *stime*$_{ij} \leftarrow \max($*vtime*$_i$, *ftime*$_{ij})$
9) *ftime*$_{ij} \leftarrow$ *stime*$_{ij} + 1.0/\phi_{ij}$
10) *valid*$_{ij} \leftarrow$ **true**
11) **if** *req*$_{ij}$ = **false**, $\forall j$ **then**
12) input i idles for the next slot

Step 2: adjust system virtual finish time.

13) *vtime*$_i \leftarrow \max($*vtime*$_i, \min($*stime*$_{ij}$ | *req*$_{ij}$ = **true**$))$

Step 3: select the output with the smallest eligible finish time.

14) $j \leftarrow \arg_j \min($*ftime*$_{ij}$ | *stime*$_{ij} \leq$ *vtime*$_i$ **and** *valid*$_{ij}$ = **true**$)$
15) *valid*$_{ij} \leftarrow$ **false**

Step 4: update system virtual time.

16) *vtime*$_i \leftarrow$ *vtime*$_i + 1.0/\sum_j \phi_{ij}$

The output arbiter is almost the same as the input arbiter, except that it checks the status of VOQs, crosspoint buffers (CBs) and assembly buffers (on the line card), and that the queue length to calculate ϕ_{ij} includes cells in both *VOQ*$_{ij}$ and *CB*$_{ij}$.

B. Remark

In theory, any PFQ algorithm works. For the sake of simple and fast hardware implementation, however, we choose WF^{2}Q+ [13] because it provides the tightest delay bound and the smallest worst-case fair index (WFI) while maintaining the lowest algorithmic complexity by computing the system virtual time directly from the packet system. WF^{2}Q+ also maintains per-flow (instead of per-packet) virtual start time and finish time, and virtual times are updated only when a packet arrives at the head of its queue, thus greatly reducing the hardware complexity.

AMFS uses the PFQ algorithms that is designed for scheduling packets at output links. However, there are still some major differences between AMFS and an output-link scheduler. First of course is the weight definition. For each flow with a fixed weight, an output-link scheduler using WF^{2}Q+ (or other PFQ algorithms) simply keeps the interval which is inverse of the weight. In addition, using normalized weights, the system virtual time can simply be increased by 1 instead of $1/\sum_j \phi_{ij}$ (in line 16). AMFS, on the other hand, has to keep track of the queue length and adjust the weight dynamically. So arithmetic reciprocal operation is needed to get the service interval (in line 9).

Second, output-link schedulers are packet driven, i.e., the algorithm executes enqueue when a packet arrives and dequeue when a packet departs. AMFS, on the other hand, is time-driven, i.e., the enqueue and dequeue operation occur at the same time in every slot because of the slotted crossbar operation. To take advantage of this synchronized operation, we also sample queue status and update virtual times only at the beginning of a slot. For cells arriving in the middle of a slot, we can regard them as arriving at the beginning of the next slot. This little postponement should have virtually no impact on the performance.

Third, output-link schedulers consider the case of N input queues and one output queue (or link). Dequeue occurs only when the output queue is available and any input queue has packets. When the output is busy, all inputs are blocked. In the crossbar scheduling, each arbiter has N input queues (e.g. VOQs) and N output queues (e.g. crosspoint buffers). When one output queue is full, only the corresponding input queue should be disabled in order to keep work-conserving. As a result, we cannot update the virtual start and finish time of this blocked queue even if it has backlogged packets. From the queuing point of view, we can regard the input queue as empty and a packet arrives at the time when its output is available. Therefore, the virtual times should not be updated until this point. That is the major purpose of counter *valid*$_{ij}$.

VIII. Simulation Results

In our simulation, we implement AMFS based on WF^{2}Q+ [13] (with $\alpha = 0.7$ and $\beta = 0.8$). WF^{2}Q+ is also used in the case of fixed weight, which is referred as PFQ in the figures in this section. We also implement OCF [20] for IQ switches because it is proved to achieve 100% throughput for any admissible traffic. Output queuing scheme with WF^{2}Q+ as the output-link scheduler is also shown for comparison as it is the optimal solution. Note in the case of IQ and CICQ switches, the output-link scheduler is simply FCFS.

The switch size is 16×16. Cell size is 64 bytes. VOQs or output queues (OQs) are statically partitioned with 256K bytes (4K cells) per input-output pair. The total buffer size per line

card for a 16×16 switch is 4M bytes. The crosspoint buffer size of CICQ switches is 8 cells unless otherwise stated.

Packet arrival is modeled as a 2-state ON-OFF process. The number of ON state slots is defined as the packet length which is obtained from a profile of NLANR trace at AIX site [25]. We collected over 119 million packets. The packet length ranges from 20 to 1500 bytes with mean $E_{on} = 566$ bytes and standard deviation of 615 bytes. The number of OFF state slots is exponentially distributed with average $E_{off} = \frac{1-\rho}{\rho}E_{on}$, where ρ is defined as the offered workload. We consider the following performance metrics:

- *average packet delay*: In the case of IQ and CICQ switches, packet delay is measured from when the first bit of a packet arrives at its VOQ to the last bit leaves its assembly buffer (see Fig. 1 and 2). In the case of OQ switches, packet delay is measured from when the first bit of a packet arrives at its OQ to the last bit leaves its OQ.
- *average queue length*: the average queue length of VOQs (in the case of IQ/CICQ switches) or OQs (in the case of OQ switches).
- *average throughput*: number of cells per slot leaving the assembly buffers (in the case of IQ/CICQ switches) or OQs (in the case of OQ switches).

The simulations run long enough to ensure the 95% confidence interval of the average packet delay, queue length or throughput with $\pm5\%$ error margin. Evaluation is performed under various traffic patterns. Due to page limit, however, we only report some of the results.

A. Admissible traffic

For uniform traffic, all schemes work well. Here we only report simulation results under the diagonal traffic: $\lambda_{ii} = 2\rho/3$, $\lambda_{i|i+1|} = \rho/3$ and $\lambda_{ij} = 0$ for $j \neq i$ or $|i+1|$, where $|i+1| = i \mod N$. This is a very skewed traffic pattern. No maximal algorithms have been found to be able to sustain admissible workload under the diagonal traffic pattern. Therefore this traffic pattern can be used as a litmus test.

Fig. 8 and Fig. 9 show the average packet delay and queue length of traffic from VOQ_{ij} as a function of the workload per input. In this situation, AMFS and OCF still perform very well. PFQ starts to drop packets at about 90% workload. To see what happened, we also plot the packet delay of traffic from $VOQ_{i|i+1|}$ in Fig. 10, where the delay is very low in the case of PFQ. This case clearly shows that favoring fairness affects the throughput adversely, as pointed out in section III.

B. Non-admissible traffic

To evaluate the weighted max-min fairness allocation, we run the simulation on a 4×4 switch with traffic weight as $W = \begin{bmatrix} 3 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. Its corresponding static max-min service rate $R^w = \begin{bmatrix} 1/2 & 1/3 & 1/6 & 0 \\ 1/2 & 0 & 0 & 0 \\ 0 & 2/3 & 0 & 0 \\ 0 & 0 & 5/6 & 0 \end{bmatrix}$. Fig. 11 shows the throughput of four schemes as a function of workload per flow. As

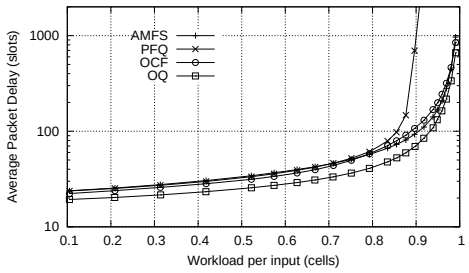

Fig. 8. Average packet delay under diagonal traffic (for VOQ_{ii}).

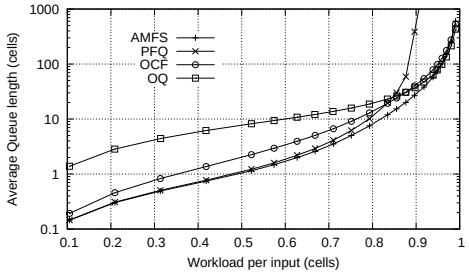

Fig. 9. Average queue length under diagonal traffic (for VOQ_{ii}).

expected, OCF cannot distinguish flows with different weights. In the case of output queuing, flows only compete for outputs. Therefore, $\frac{\mu_{11}}{\mu_{21}} = \frac{3}{1}$, $\frac{\mu_{12}}{\mu_{32}} = \frac{2}{1}$ and $\frac{\mu_{13}}{\mu_{43}} = \frac{1}{1}$ when outputs are fully overloaded.

On the other hand, AMFS and PFQ always maintain max-min allocation among competing flows. When the workload is below $1/3$, the throughput is equal to the workload for every flow because the switch is not overloaded.

At 0.33, input 1 is first saturated, the throughput of f_{13} goes down until reaching its max-min rate $1/6$. For f_{12}, although $\lambda_{12} > r^w_{12} = 1/3$ when $1/3 < \lambda < 2/5$, it is still an admissible flow because $\lambda_{12} \leq r^{w\lambda}_{12}$, After 0.40, the throughput of f_{12} also starts to go down until it reaches its max-min rate.

At workload 0.5, output 1 is also saturated, and f_{11} gets its max-min rate $1/2$. Note that although the weight of f_{21} is 1 (corresponding to $1/6$ of the total bandwidth), it also receives $1/2$ of the total bandwidth because of the max-min fair allocation policy.

Finally when output 2 and output 3 become saturated at $2/3$ and $5/6$, f_{32} and f_{43} receive their max-min fair shares

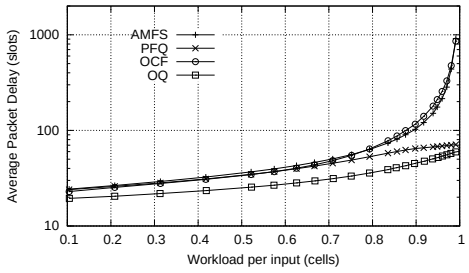

Fig. 10. Average packet delay under diagonal traffic (for $VOQ_{i|i+1|}$).

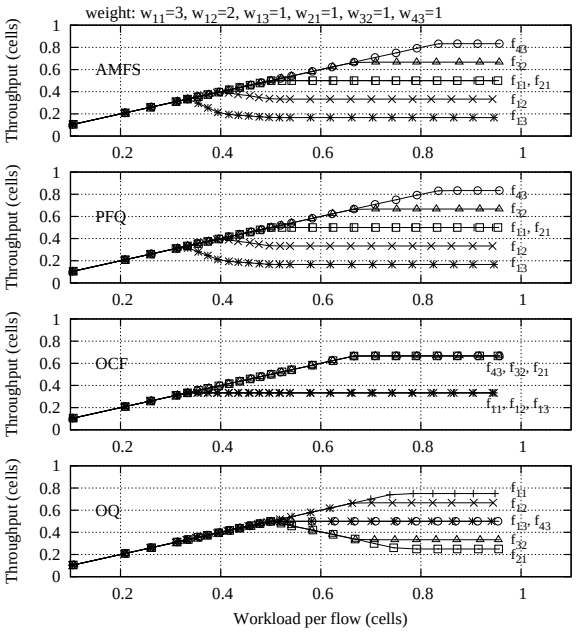

Fig. 11. Throughput under non-admissible traffic.

respectively which are also much more than their assigned bandwidth.

IX. Hardware Implementation Issues

Due to page limit, we only show in Fig. 12 a block diagram of how four steps of an AMFS arbiter are connected, and omit detailed discussion of each block. Each arbiter is basically a WF^{2}Q+ arbiter [13] with dynamic weight. With parallel hardware support, each step can be done in $O(\log N)$ time.

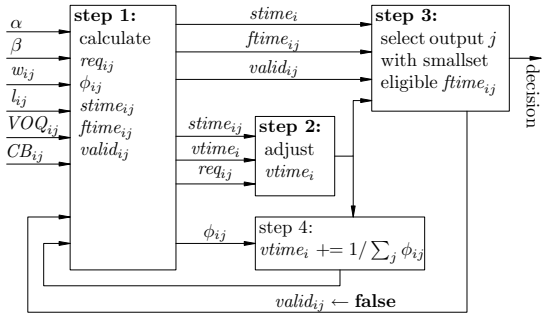

Fig. 12. A block diagram of an AMFS arbiter at input i

X. Conclusion

In this paper, we address the problem of how to achieve both throughput and fairness in a buffered crossbar without speedup. The solution is surprisingly simple: applying a PFQ algorithm at each input and output with the dynamic weights based on queue lengths and assigned weights. Our analysis and simulation results show that the adaptive max-min fair scheduling (AMFS) scheme achieves 100% throughput for any admissible traffic as well as providing max-min fairness under overloaded situation. We also briefly show the hardware implementation of AMFS. With each arbiter on its line card, AMFS is entirely feasible for very high speed networks.

References

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

[2] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Networking*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[3] N. Ni and L. N. Bhuyan, "Fair scheduling and buffer management in internet routers," in *Proc. IEEE INFOCOM'02*, vol. 3, Yew York, June 2002, pp. 1141–1150.

[4] X. Zhang and L. N. Bhuyan, "Deficit round-robin scheduling for input-queued switches," *IEEE J. Select. Areas Commun.*, vol. 21, no. 4, pp. 584–594, May 2003.

[5] J. G. Dai and B. Prabhakar, "The throughput of data switches with and without speedup," in *IEEE INFOCOM*, vol. 3, Mar. 2000, pp. 556–564.

[6] M. Nabeshima, "Performance evaluation of a combined input- and crosspoint-queued switch," *IEICE Trans. Commun.*, vol. E83-B, no. 3, pp. 737–741, Mar. 2000.

[7] R. Rojas-Cessa, E. Oki, Z. Jing, and H. J. Chao, "CIXB-1: combined input-one-cell-crosspoint buffered switch," in *Proc. IEEE HPSR'01*, May 2001, pp. 324–329.

[8] T. Javidi, R. Magill, and T. Hrabik, "A high-throughput scheduling algorithm for a buffered crossbar switch fabric," in *Proc. IEEE ICC'01*, June 2001, pp. 1581–1587.

[9] X. Zhang and L. N. Bhuyan, "An efficient algorithm for combined-input-crosspoint-queued (CICQ) switches," in *Proc. IEEE GLOBECOM'04*, vol. 2, Nov./Dec. 2004, pp. 1168–1173.

[10] A. Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Networking*, vol. 1, no. 3, pp. 344–357, June 1993.

[11] A. Demers, S. Keshav, and S. Shenkar, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, Sept. 1989.

[12] J. C. Bennett and H. Zhang, "WF^{2}Q: Worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM'96*, Mar. 1996, pp. 120–128.

[13] ——, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Trans. Networking*, vol. 5, no. 5, pp. 675–689, Oct. 1997.

[14] D. C. Stephens and H. Zhang, "Implementing distributed packet fair queuing in a scalable switch architecture," in *Proc. IEEE INFOCOM'98*, vol. 1, Mar. 1998, pp. 282–290.

[15] R. B. Magill, C. E. Rohrs, and R. L. Stevenson, "Output-queued switch emulation by fabrics with limited memory," *IEEE J. Select. Areas Commun.*, vol. 21, no. 4, May 2003.

[16] S.-T. Chuang, S. Iyer, and N. McKeown, "Practical algorithms for performance guarantees in buffered crossbars," in *Proc. of IEEE INFOCOM'05*, Miami, Florida, Mar. 2005.

[17] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input versus output queuing on a space-division packet switch," *IEEE Transaction on Communications*, vol. COM-35, no. 12, pp. 1347–1356, December 1987.

[18] 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.

[19] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE J. Select. Areas Commun.*, vol. 17, no. 6, pp. 1030–1039, June 1999.

[20] A. Mekkittikul and N. McKeown, "A starvation-free algorithm for achieving 100% throughput in an input-queued switch," in *ICCCN'96*, Oct. 1996, pp. 226–231.

[21] D. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall, 1992.

[22] N. Chrysos and M. Katevenis, "Weighted fairness in buffered crossbar scheduling," in *Proc. IEEE HPSR'03*, June 2003.

[23] G. F. Georgakopoulos, "Nash equilibria as a fundamental issue concerning network-switches design," in *Proc. IEEE ICC'04*, vol. 2, June 2004.

[24] E. Leonardi, M. Mellia, F. Neri, and M. A. Marsan, "On the stability of input-queued switches with speed-up," *IEEE/ACM Trans. Networking*, vol. 9, no. 1, pp. 104–118, Feb. 2001.

[25] National Laboratory for Applied Network Research, "NLANR network traffic packet header traces." [Online]. Available: http://pma.nlanr.net/Traces