# Design, Analysis, and Implementation of a Novel Low Complexity Scheduler for Joint Resource Allocation

Fariza Sabrina, *Member*, *IEEE*, Salil S. Kanhere, *Member*, *IEEE*, and Sanjay K. Jha, *Member*, *IEEE*

**Abstract**—Over the past decade, the problem of fair bandwidth allocation among contending traffic flows on a link has been extensively researched. However, as these flows traverse a computer network, they share different kinds of resources (e.g., links, buffers, router CPU). The ultimate goal should hence be overall fairness in the allocation of multiple resources rather than a specific resource. Moreover, conventional resource scheduling algorithms depend strongly upon the assumption of prior knowledge of network parameters and cannot handle variations or lack of information about these parameters. In this paper, we present a novel scheduler called the *Composite Bandwidth and CPU Scheduler (CBCS)*, which jointly allocates the fair share of the link bandwidth as well as processing resource to all competing flows. CBCS also uses a simple and adaptive online prediction scheme for reliably estimating the processing times of the incoming data packets. Analytically, we prove that CBCS is efficient, with a per-packet work complexity of $O(1)$. Finally, we present simulation results and experimental outcomes from a real-world implementation of CBCS on an Intel IXP 2400 network processor. Our results highlight the improved performance achieved by CBCS and demonstrate the ease with which it can be implemented on off-the-shelf hardware.

**Index Terms**—Packet-switched networks, distributed applications, microprocessor/microcomputer applications, scheduling.

## 1 Introduction

Fair allocation of shared network resources among multiple users is an intuitively desirable property. Strict fairness in traffic scheduling improves the isolation between flows, offers a more predictable performance, and eliminates bottlenecks. Fair resource allocation also plays an important part in Quality of Service (QoS) mechanisms that ensure end-to-end performance guarantees such as minimum bandwidth and delay bounds.

The link bandwidth, however, is not the only resource that is shared by the traffic flows as they traverse the network. A router's processor is often also a critical resource to which all competing flows should have fair access. For each incoming packet, the router has to perform several activities which may include computing the checksum, performing a forwarding table lookup, processing variable length options, etc. Given the fact that processing requirements of different packets vary widely, the issue of fairness in the allocation of the processing resources gains significance. Besides the fact that packet lengths can vary widely, there is also a wide variation in the ratio of a packet's demand for bandwidth and its demand for processing cycles. This is primarily due to the different kinds of control data carried by individual packets, for example, in the optional headers. Thus, one cannot achieve overall fairness merely by fair sharing of the link bandwidth alone or merely through fair allocation of the processing resource alone. Moreover, the allocation of bandwidth and CPU resources are interdependent and the fair allocation of one resource does not necessarily entail fairness in the other resource allocation. Therefore, for better maintenance of QoS guarantees and overall fairness in resource allocations for the contending flows, it is vital that the processor and bandwidth scheduling schemes should be integrated. This is particularly important, given the extensive packet processing capabilities of today's network processors, e.g., Intel IXP2400 [1].

- *F. Sabrina is with the CSIRO ICT Centre, Cnr Vimiera and Pembroke Roads, Marsfield, NSW 2122, Australia. E-mail: Fariza.Sabrina@csiro.au, farizas@gmail.com.*
- *S.S. Kanhere and S.K. Jha are with the School of Computer Science and Engineering, University of New South Wales, Sydney NSW 2052, Australia. E-mail: {salilk, sjha}@cse.unsw.edu.au.*

*Manuscript received 8 Dec. 2005; revised 28 Aug. 2006; accepted 18 Sept. 2006; published online 9 Jan. 2007.*
*Recommended for acceptance by X. Zhang.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0501-1205.*
*Digital Object Identifier no. 10.1109/TPDS.2007.1045.*

### 1.1 Related Work

Generalized Processor Sharing (GPS) [2] is an ideal but unimplementable scheduling discipline for fair allocation of a single resource. During the past several years, a variety of algorithms have been proposed for achieving fair allocation of the link bandwidth among multiple flows that share the link [13], [3]. However, most of these schemes cannot be readily adapted for processor scheduling because they require precise knowledge of the execution times for the incoming packets at the time of their arrival in the node. Start-Time Fair Queuing (SFQ) [4], a bandwidth scheduling algorithm that does not require knowledge of the packet length prior to making a scheduling decision, could be a suitable option for scheduling computational resources. However, the worst-case delay under SFQ increases with the number of flows. Further, it tends to favor flows that have a higher average ratio of processing time per packet to reserved processing rate [5]. Over the years, several

1045-9219/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

researchers working in the field of computer architecture have proposed schemes for scheduling access to the CPU processing cycles [6], [7], but most of these proposals focus on CPU scheduling for end systems and work at the task level, not at the packet level.

Pappu and Wolf [5] presented a processor scheduling algorithm for programmable routers called Estimation-based Fair Queuing (EFQ) that estimated the execution times of various applications on packets of given lengths offline and then scheduled the processing resources based on the estimations. Fixed values of the estimation parameters measured offline may not always produce good estimations due to variation in server load and operating system scheduling. Galtier et al. [8] proposed a scheme to predict the CPU requirements of executing a specific code on a variety of platforms. However, the scheme seems too complicated to be implemented with programmable routers.

A few scheduling schemes have also been proposed for multiprocessors, but these are mostly simple static policies [9], [10]. Shi et al. [11] describe the design and implementation of the Dynamic Window-Constrained Scheduling (DWCS) algorithm for scheduling packets on network processors. However, DWCS was developed only for bandwidth scheduling. Researchers in [12] developed another scheduler called the Packet Timed Token Service Discipline (PTTSD), which allows the coexistence of guaranteed service flows such as real-time connections and classic best-effort flows and schedules bandwidth resources among the competing flows.

Most existing scheduling disciplines can be broadly classified into the following categories:

- Sorted Priority Schedulers: These schedulers maintain a global variable known as *virtual time*. A timestamp computed as a function of this variable is associated with each packet in the system. Packets are scheduled in the increasing order of these timestamps. Examples include WFQ [13], Self-Clocked Fair Queuing (SCFQ) [14], and SFQ [4]. The main disadvantage of these schemes is that the per-packet work complexity is a function of the total number of flows being served by the scheduler, which makes them hard to implement in high-speed hardware.
- Round Robin Schedulers: In these schemes (i.e., Deficit Round Robin (DRR) [15], Elastic Round Robin (ERR) [16], etc.), on the other hand, the scheduler provides service opportunities to the backlogged flows in a round-robin order and, during each service opportunity, the intent is to provide the flow with an amount of service proportional to its fair share of the resource. Since these schemes do not require any sorting among the packets, their implementation complexity is $O(1)$, making them attractive for implementation in high-speed routers.

All the scheduling schemes discussed above are designed to schedule only a single resource, i.e., either bandwidth or processing resource. Although the determination of execution times for packets in advance on a programmable node has been identified as a major obstacle in managing processing resources [5], [8], [17], none of the previous studies provided a generalized online solution to the problem.

### 1.2 Contributions

The main contributions of this paper can be summarized as follows:

- We present a composite bandwidth and processor scheduler called *Composite Bandwidth and CPU Scheduler (CBCS)*, which can schedule multiple resources adaptively, fairly, and efficiently among all the competing flows. Our scheduler employs a simple and adaptive online prediction scheme called modified SES for determining the packet execution times. CBCS has a per-packet work complexity of $O(1)$, making it attractive for implementation in high-speed routers.
- This paper also analytically derives the relative fairness bound (FB) of CBCS, a popular metric for evaluating the fairness properties. Finally, the worst-case work complexity, a metric that highlights the efficiency of CBCS, is evaluated. Our analysis shows that CBCS has better fairness characteristics and a significantly lower latency bound in comparison to separate CPU and bandwidth schedulers. Simulation-based evaluations to confirm our analytical results are also presented.
- The FB is a fairness measure which highlights the worst-case performance of the scheduler. It, however, does not accurately capture the behavior of the scheduler under normal circumstances. We use a metric called Gini Index, borrowed from the field of economics, to comparatively judge the instantaneous fairness achieved by CBCS. Simulation results to demonstrate the improved instantaneous fairness of CBCS are also presented.
- With the rapid increase in the capacity of transmission links, the ease with which a scheduler can be implemented in real hardware systems is extremely crucial. We demonstrate that our scheduler can be easily implemented on an off-the-shelf network processor such as the Intel IXP 2400 board. The extensive experimental results from the IXP2400 implementation highlight the effectiveness and high performance of this algorithm in a real-world system. To the best of our knowledge, this is one of the first public domain implementations of such a system and we expect that this will generate a lot of interest in the research community and open avenues for future research.

### 1.3 Organization

The rest of the document is organized as follows: Section 2 briefly describes the CBCS scheduling algorithm and also discusses the prediction scheme used for estimating the processing duration of the packets. Section 3 presents analytical results on the efficiency and fairness characteristics of CBCS. Simulation results demonstrating the improved performance of PCFQ over a system consisting of separate bandwidth and CPU schedulers are presented in Section 4. Section 5 discusses the details of the implementation of CBCS on the Intel IXP2400 network processor and outlines the experimental results. Finally, Section 6 concludes the paper.

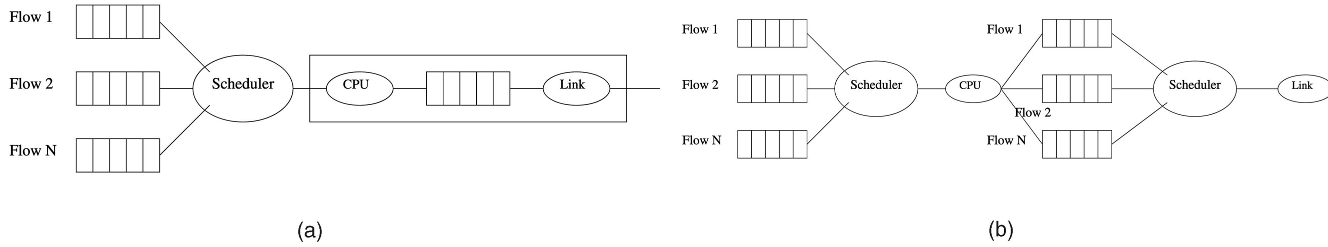



Fig. 1. System model for CBCS and separate processor and BW scheduler. (a) CBCS. (b) Separate processor and BW scheduler.

## 2 CBCS—Composite Bandwidth and CPU Scheduling Algorithm

In this section, we present a brief overview of the CBCS scheduler and prediction technique used to estimate the packet processing duration. A more detailed description can be found in [18], [19], and [20].

### 2.1 Online Prediction Process

Since the processing requirement of each packet is not known a priori, the CBCS scheduler needs to estimate the processing duration for each arriving packet. We have investigated several smoothing methods and their suitability for predicting the processing requirements of the packets. A detailed analysis, investigations, and a comparative performance analysis of the alternatives are discussed in [20]. Our investigations show that the Single Exponential Smoothing (SES) technique is well-suited to estimate the execution times of the packets. SES is computationally simple and an attractive method of forecasting. Researchers have used this method to forecast the display cycle time (which includes decompression time plus rendering time) for compressed video data packets [21]. SES uses the following equation to calculate a new predicted value:

$$F_{t+1} = \alpha X_t + (1 - \alpha)F_t, \qquad where\ 0 \leq \alpha \leq 1, \tag{1}$$

where $F_t$ and $F_{t+1}$ are the predicted value at $t$th and $(t+1)$ periods, respectively. $X_t$ is the actual duration required to process the process the packet that arrived at time $t$ and $\alpha$ is the SES coefficient which determines the relative weight allocated to the history and the current estimated sample.

Most of the packets that are processed by today's routers can be broadly classified into two categories based on their processing needs: 1) header processing and 2) payload processing. A header processing application (such as IP forwarding, transport layer classification, or QoS routing) only requires read and write operations in the header of the packet and, so, the processing complexity is, in general, independent of the size of the packets. In contrast, a payload processing application (such as IPSec Encryption, packet compression and packet content transcoding (e.g., image format transcoding), or aggregation of sensor data, etc.) involves read and write operations on all the data in the packet, in particular, the payload of the packet, and, therefore, the processing complexity strongly correlates to the packet size [5]. In order to account for the correlation between the processing costs and packet sizes, we define a parameter called the *Scaling Factor (SF)* as $SF = 1$ for header processing packets and $\frac{L_{t+1}}{L_t}$ for payload processing packets.

Here, $L_t, L_{t+1}$ = Length of the packet arriving at time $t$ and $(t+1)$, respectively.

The scaling factor is incorporated in the SES estimation as follows:

$$F_{t+1} = SF\{\alpha X_t + (1 - \alpha)F_t\}, \qquad where\ 0 \leq \alpha \leq 1. \tag{2}$$

### 2.2 Overview of the CBCS Algorithm

We first begin by briefly describing the system model used in our work. As shown in Fig. 1a, a set of $N$ flows share a processor and a link. Packets from each flow are first processed by the processor and then transmitted onto the output link. The joint allocation of the processing and bandwidth resource is accomplished by the composite scheduler which selects a packet from the input buffers and passes it onto the CPU for processing. No scheduling action takes place after the processing; the packets processed by the CPU are stored in the buffer between the processor and the link and are transmitted in a first-come-first serve order. The CBCS scheduler is based on the principles used in DRR [15]. Further, contrary to the single resource schedulers, CBCS is designed to schedule both bandwidth and CPU resources adaptively, fairly, and efficiently among all the competing flows. It succeeds in eliminating the unfairness of pure packet-based round-robin by maintaining a *Credit Counter* to measure the past unfairness. *Credit Counter* is similar to the variables *Deficit counter* and *Surplus count* used by single resource schedulers such as DRR [15] and ERR [16], respectively. All backlogged flows are stored in a linked list and the flows are served in a round-robin order by the CBCS scheduler. The algorithm uses the following parameters and equations:

- $Quantum$: A variable that represents a time slice used to serve packets from each flow queue, which includes both CPU processing time and network transmission time (in msec). Let $Q(r)$ denote the quantum in round $r$.
- $CC[i]$: Credit Counter, a state variable that represents a time slice for which flow $i$ deserves to be served within a specific round of scheduling (in msec). Let $CC_r[i](r)$ represent the Credit Counter for flow $i$ in round $r$.
- $BW$: Bandwidth of the transmission link in Mbps.
- $EP_i^k$: Estimated processing cost of packet $k$ of flow $i$ in seconds.
- $L_i^k$: Length of packet $k$ of flow $i$ in bits.
- $\gamma^i$: Resize factor of the packets in flow $i$.

- $c_t^{UL}$, $c_t^{LL}$: Upper and lower limit (respectively) of the total CPU queue in terms of the CPU processing time requirement for all the packets in all the flows.
- $Ptc_i^k$: The combined processing and transmission cost for the $k\text{th}$ packet of flow $i$.
- $Ptc^{Max}$: The maximum allowable time slice that a packet requires to cover both CPU processing and network transmission among all flows (total cost).

Therefore,

$$Ptc_i^k = \frac{1{,}000 * \gamma^i * L_i^k}{BW} + EP_i^k. \quad (3)$$

On receiving a new packet, the scheduler examines the header to determine the flow-id, calculates the CPU processing time using the online prediction scheme discussed earlier and the resize factor, and then stores the packet in the corresponding flow queue. The CBCS scheduler continues to monitor the queue length for all individual flows in terms of the CPU time requirement and stops accepting packets from a flow if its queue length becomes greater than $c_{t[i]}^{UL}$. In this case, the scheduler continues to refuse new packets from flow $i$ until its queue length becomes smaller than $c_{t[i]}^{LL}$.

Upon initialization, the $Quantum$ is set to $Ptc^{Max}$ and the Credit Counter $(CC[i])$ for all flows are set to zero. The scheduler continues to serve all nonempty queues within each round of processing. When it starts to serve a queue within a round, the Credit Counter is set to $Quantum$ plus the Credit Counter of the previous round. The scheduler then dequeues a packet from the head of the queue and calculates the $Ptc_i^k$ of the packet according to the (3). It sets the $CC[i]$ to $(CC[i] - Ptc_i^k)$ and hands the packet to the processor Handler object for execution. The packet is sent to its next destination after processing. The scheduler stops serving a queue once the queue is empty or the credit counter becomes zero or negative. It may be noted that the $CC[i]$ for a nonactive flow (i.e., a flow having no packets in the queue) is reset to zero.

It should be mentioned that CBCS can be easily adapted for scheduling a guaranteed rate connection by assigning weight to each flow. Let $\phi_c^i$ and $\phi_b^i$ be the weights for CPU and bandwidth for flow $i$. If $r_c^i$ and $r_b^i$ are the CPU and bandwidth reservation for flow $i$, respectively, and $r_c^m$ and $r_b^m$ are the minimum CPU and bandwidth reservation among all flows, respectively, the weights are assigned as below: $\phi_c^i = \frac{r_c^i}{r_c^m}$ and $\phi_b^i = \frac{r_b^i}{r_b^m}$. The weighted version of CBCS is exactly similar to the CBCS algorithm described in the preceding section. The only difference is in the calculation of the $Quantum$. If $w^i$ is the summation of the weights (both CPU and bandwidth) of flow $i$ and $w^m$ is the smallest summation of the weights among all flows, the $Quantum$ of each flow is calculated as $\frac{w^i}{w^m} * Ptc^{Max}$.

# 3 Analytical Results

In this section, we analytically derive the work complexity and fairness properties of CBCS. We also compare the behavior of CBCS to a typical implementation found in most existing routers today, wherein separate schedulers are responsible for the scheduling bandwidth and the CPU. The exact scheduling algorithm used in each scheduler can be chosen from the host of available scheduling disciplines. In our comparisons, we pick DRR [15], the most popular round robin scheduler, as a representative example. Note that DRR cannot be easily adapted to schedule the processing resource since it requires knowledge of the processing duration of each packet. However, for this comparison, we assume that the exact execution durations for each packet are precisely available. The system model for such a CPU and bandwidth scheduler is illustrated in Fig. 1b.

## 3.1 Work Complexity

The work complexity of a scheduler is defined as the order of time complexity with respect to enqueuing and then dequeuing a packet for transmission.

**Theorem 1.** *The worst-case work complexity of the CBCS scheduler is $O(1)$.*

**Proof.** The enqueue operation consists of determining the flow at which the packet arrives and adding the flow to the linked list if it is not already in the list. Both of these operations are $O(1)$. The dequeue procedure involves determining the next flow to be served, calculating the credit counter, and removing the flow from the active list. All of these can be done in constant time, so we can say that dequeue operation is of time complexity $O(1)$. As the complexity of both the enqueuing and dequeuing tasks is $O(1)$, it follows that the work complexity of the CBCS scheduler is $O(1)$. □

## 3.2 Fairness Bound

In our fairness analysis, we use the popular metric, *Relative Fairness Bound (RFB)*, first proposed in [14]. The RFB is defined as the maximum difference in the service received by any two flows over all possible intervals of time. The total resource consumed by a flow, i.e., the sum of the processing and bandwidth resources, is used as the basis for measuring fairness. We first introduce a few notations and definitions which will be used in the rest of the analysis. Let us assume $Pcc_i(t_2 - t_1)$ and $Pbc_i(t_2 - t_1)$ are the CPU and transmission time consumed by the packets in flow $i$ within a time period of $(t_2 - t_1)$, respectively, and $Ptc^{Max}$, $Ptc_i^{Max}$, and $Ptc_j^{Max}$ are the maximum total cost (in terms of ms) that a packet (among all flows, within $i\text{th}$ and $j\text{th}$ flow, respectively) requires to cover both CPU processing and network transmission.

**Definition 1.** *A flow is backlogged during the time interval $(t_2 - t_1)$ if the queue for the flow is never empty during the interval.*

**Definition 2.** *Using the notation above, for any period of time interval $(t_2 - t_1)$, the total resource allocated to flow $i$ is given by*

$$Ptc_i(r) = Pcc_i(t_2 - t_1) + Pbc_i(t_2 - t_1).$$

**Definition 3.** *Let $R_i(t_1, t_2)$ be the total resource, measured in units of time, received by flow $i$ during the time interval between $t_1$ and $t_2$. For an interval $(t_1, t_2)$, the Relative Fairness, $RF(t_1, t_2)$, is defined as the maximum value of $|R_i(t_1, t_2) - R_j(t_1, t_2)|$ over all pairs of flows $i$ and $j$ that are active during this interval. The Relative Fairness Bound (RFB) is defined as the maximum of $RF(t_1, t_2)$ over all possible time intervals $(t_1, t_2)$.*

To evaluate the RFB for the CBCS scheduler, we first present the following lemma which determines the upper and lower bounds for the credit counter:

**Lemma 1.** *For any flow $i$ and round $r$,*

$$-\left(Ptc_i^{max}(r) - \delta\right) \leq CC[i](r) \leq Q(r). \quad (4)$$

**Proof.** As was described earlier, resource is allocated to flows as long as the credit counter (which is allocated resource − used resource) is $\geq 0$, and the credit counter is calculated after each time a flow is served, so the new value of $CC[i](r)$ would be based on the present resource usage. For the worst case, we assume the value of the credit counter is slightly more than zero in the case when the lower bound is $\delta - Ptc_i(r)$. If the packet's total cost (processing and transmission) is highest among all the packets within the flow, the credit counter could be $\delta - Ptc_i^{max}(r)$. Here, $\delta$ is a very small amount. If the flow becomes empty, then the credit counter is reset to 0. □

Next, we derive the bounds on the service received by any flow during $n$ consecutive rounds of execution.

**Theorem 2.** *For $n$ consecutive rounds starting from $k$, during which flow $i$ was active, the bounds on the total resources allocated to flow $i$ are given by*

$$\sum_{r=k}^{k+n} Ptc_i(r) - Ptc_i^{max} \leq R_i(r) \leq \sum_{r=k}^{k+n} P_i(r) + Ptc_i^{max}.$$

**Proof.** Our approach here is similar to the one used in [22]. Let us assume that $AR[i](r)$ and $UR[i](r)$ represent the allocated and used resource, respectively, for flow $i$ in round $r$. Hence,

$$AR[i](r) = Q(r) + CC[i](r), \quad (5)$$

$$CC[i](r+1) = AR[i](r) - UR[i](r). \quad (6)$$

From (6), we have

$$UR[i](r) = AR[i](r) - CC[i](r+1), \quad (7)$$

$$UR[i](r) = Q(r) + CC[i](r) - CC[i](r+1). \quad (8)$$

The maximum value of the quantum should be the maximum resource that could be required by any packet of any flow. Summing the LHS in (8) for $r = k$ to $r = k + n - 1$, we get $R_i(r)$, the total amount of combined resource (both CPU and BW, measured in units of time) by flow $i$ during the $n$ consecutive rounds under consideration. Equating the summation for $r = k$ to $r = k + n - 1$, we get

$$UR_i^n = \sum_{r=k}^{k+n-1} Q(r) + CC[i](k) - CC[i](K+n). \quad (9)$$

Using Lemma 1, we prove the theorem. □

We now present a lemma which allows us to pick specific time intervals for evaluating the RFB.

**Lemma 2.**

$$RFB = \max_{t_1, t_2 \in T_s} RF(t_1, t_2).$$

**Proof.** Here, $T_s$ is the set of all time instants at which the scheduler ends serving one flow and begins serving another, and $T$ is the set of all time instants during an execution of the $CBCS$ algorithm. We could prove the lemma if, for any $t_1$, $t_2 \in T$, there is $t_1'$, $t_2' \in T_s$, such that $RF(t_1', t_2') \geq RF(t_1, t_2)$.

If there are two active flows $i$ and $j$ during the interval between $t_1$ and $t_2$, where $t_1$, $t_2 \in T$, let us assume that, during this interval, flow $i$ got more service than flow $j$. By appropriately choosing $t_1'$ as the time instant at either the begining or the end of the service opportunity given to a flow at time $t_1$, one may verify that $RF(t_1', t_2) \geq RF(t_1, t_2)$. Similarly, $t_2'$ could be chosen as either the begining or the ending instant of the service opportunity given to a flow at $t_2$ so that $RF(t_1', t_2') \geq RF(t_1, t_2)$. □

**Theorem 3.** *For any execution of the CBCS discipline, $RFB \leq 3Ptc^{max}$.*

**Proof.** Our approach is similar to that used in [22]. Let us consider two flows $i$ and $j$ that are active in the time interval between $t_1$ and $t_2$. The CBCS algorithm states that, after any flow receives service, it is added to the tail end of the *ActiveList*. So, after flow $i$ is served, the scheduler serves flow $j$ before flow $i$ receives the service. Thus, in between two consecutive service opportunities given to flow $i$, flow $j$ receives exactly one service opportunity. If $n_i$ and $n_j$ denote the total number of rounds received by flow $i$ and $j$, respectively, in the time interval $(t_1, t_2)$, then $|n_i - n_j| \leq 1$.

Let $r(t)$ denote the round in progress at time instant $t$. Note that the time instant $t_1$ may be such that the service opportunity received by one of the two flows in round $r(t_1)$ may not be a part of interval $(t_1, t_2)$. Thus, the first time that the scheduler visits this flow in the interval under consideration would be in the round following $r(t_1)$. Consequently, if $r_i$ and $r_j$ denote the rounds in which flows $i$ and $j$ receive service for the first time in the interval $(t_1, t_2)$, respectively, then $|r_i - r_j| \leq 1$. Let us assume that, within the interval $(t_1, t_2)$, flow $i$ starts receiving service before flow $j$. Thus,

$$r_j \leq r_i + 1, \qquad n_i \leq n_j + 1.$$

From Theorem 2, for flow $i$,

$$R_i(t_1, t_2) \leq \sum_{r=k}^{k+n} Q(r) + Ptc_i^{max}, \quad (10)$$

$$\sum_{r=k}^{k+n} Q(r) - Ptc_j^{max} \leq R_j(t_1, t_2). \quad (11)$$

Combining (10) and (11) and using (10), we get

$$\begin{aligned} R_i(t_1, t_2) - R_j(t_1, t_2) \leq\ & Ptc_i^{max} \\ & + Ptc_j^{max} + \sum_{k=r_i}^{r_i+n_i} Q(k) - \sum_{k=r_j}^{r_j+n_j} Q(k). \end{aligned} \quad (12)$$

Let us consider the quantity $G$ given by

$$G = \sum_{k=r_i}^{r_i+n_i} Q(k) - \sum_{k=r_j}^{r_j+n_j} Q(k). \quad (13)$$

We now compute $G$ for each of the four possible cases.

*Case 1* ($r_i = r_j, n_i = n_j$): From (13), we have

$$G = \sum_{k=r_i}^{r_i+n_i} Q(k) - \sum_{k=r_i}^{r_i+n_i} Q(k) \Rightarrow G = 0. \tag{14}$$

*Case 2* ($r_i = r_j, n_i = n_j + 1$): From (13), we have

$$\begin{aligned} G &= \sum_{k=r_i}^{r_i+n_i} Ptc^{max}(k) - \sum_{k=r_i}^{r_i+n_i-1} Ptc^{max}(k) \\ &\Rightarrow G = Q(r_i + n_i) \\ &\Rightarrow G = Ptc^{max}(r_i + n_i). \end{aligned} \tag{15}$$

*Case 3* ($r_i = r_j - 1, n_i = n_j$):

$$\begin{aligned} G &= \sum_{k=r_i}^{r_i+n_i} Ptc^{max}(k) - \sum_{k=r_i+1}^{r_i+1+n_i} Ptc^{max}(k) \\ &\Rightarrow G = Q(r_i) - Q(r_i + n_i + 1) \\ &\Rightarrow G = Ptc^{max}(r_i) - Ptc^{max}(r_i + n_i + 1). \end{aligned} \tag{16}$$

*Case 4* ($r_i = r_j - 1, n_i = n_j + 1$):

$$\begin{aligned} G = \sum_{k=r_i}^{r_i+n_i} Ptc^{max}(k) - \sum_{k=r_i+1}^{r_i+n_i} Ptc^{max}(k) \Rightarrow G = Q(r_i) \\ \Rightarrow G = Ptc^{max}(r_i). \end{aligned}$$

For all the above cases, $G \leq Ptc^{max}$. Substituting the value of $G$ in (12), we can conclude that

$$RF \leq Ptc_i^{max} + Ptc_j^{max} + Ptc^{max}.$$

In the worst case, $Ptc_i^{max} = Ptc_j^{max} = Ptc^{max}$. This proves the theorem. □

We now evaluate the RFB for a system consisting of two concatenated schedulers, one responsible for the CPU scheduling and other for the bandwidth, as illustrated in Fig. 1b. For a fair comparison, we pick DRR [15], a popular $O(1)$ scheduler. Let $RFB_{sep}$ and $RFB_{cbcs}$ refer to the RFB for a system consisting of concatenated DRR schedulers and the CBCS scheduler, respectively.

**Theorem 4.** $RFB_{sep} \geq RFB_{cbcs}$.

**Proof.** From Theorem 3, we see that, for CBCS, $RFB_{cbcs} \leq 3Ptc^{max}$. Now, if the CPU and bandwidth were scheduled separately, the *Relative Fairness Bound* would be $RFB_{sep} \leq 3(Ptc_c^{max} + Ptc_b^{max})$, where $Ptc_c^{max}$ and $Ptc_b^{max}$ are the maximum processing and transmission cost, respectively, for a single resource DRR scheduler. In the worst-case scenario, a packet with the highest processing cost would also require the longest transmission time, resulting in a situation where $RFB_{cbcs} = RFBsep$. However, in most situations, these two packets will actually be different. In these situations, $Ptc^{max} \leq 3(Ptc_c^{max} + Ptc_b^{max})$ and, thus, $RFB_{cbcs} \leq RFB_{sep}$. □

TABLE 1
Settings for Individual Flows

| Flow Number | 1-10 | 11-20 | 21-30 |
|---|---|---|---|
| Referenced Application | MPEG2 Encoder | RC2 Decryption | RC2 Encryption |
| Data Size | 1.5-24 KB | 4-16 KB | 1-8 KB |
| CPU Requirements per data block | 10-40msec | 1-3 msec | 1-3 msec |
| Resize Factor | 0.11 -0.36 | 0.25 | 4.0 |

## 4 Simulation Results

In this section, we present simulation results on the delay characteristics and the fairness properties of the CBCS scheduler. The simulations were performed using the ns-2 network simulator [23] on a 1.8-GHz Pentium 4 PC with 384 MB memory running the Redhat 7.2 Linux operating system. Our simulation model consists of 30 UDP flows sharing a single processor and a link. The simulation settings of the individual flows are given in Table 1. The output link capacity was set to 10 Mbps. The simulations were run for 300 seconds and samples were collected at 3 second intervals. The packet generation rates for all the flows were adjusted such that the cumulative demand for the CPU and bandwidth resources were 97 percent and 96 percent, respectively. This ensures that the measured delays reflect the performance of the scheduler and are not affected by large queuing delays. We compare the performance of CBCS with an implementation consisting of separate DRR schedulers for CPU and bandwidth scheduling. We assume that the DRR CPU scheduler uses the same online prediction scheme SES, used by CBCS for estimating the processing duration of each packet.

### 4.1 Delay Measurement

Table 2 shows the maximum delay, average delay, and the standard deviation of the delays for all three type of flows using CBCS and separate DRR schedulers. It also shows the performance improvement achieved using CBCS.

### 4.2 Packet Loss

We evaluated the packet loss characteristics of CBCS by simulating a highly congested network where the CPU and bandwidth requirements are greater than their respective capacities. The comparison of the observed packet loss in CBCS to that with separate DRR schedulers for bandwidth and CPU is shown in Table 3 (due to space limitation, we only show sample results for five flows). With DRR, a significant number of packets from the RC2 encryption flows (flows 21 to 30) were dropped after being processed, thus wasting the consumed CPU resources. RC2 encryption requires low CPU cycles but higher bandwidth since, after being encrypted, the packets become bigger. When separate DRR schedulers are used, the CPU scheduler processes a large number of packets and does achieve fairness. However, the high bandwidth demand overloads the bandwidth scheduler, which results in a large number of packets being dropped for the RC2 encryption flows.

TABLE 2
Delay Measurements

| Data | Delay using CBCS(sec) | | | Delay using DRR(sec) | | | Delay Reduced using CBCS | |
|---|---|---|---|---|---|---|---|---|
| flow | Max. del. | Avg. del. | std. dev. | Max.del. | Avg. del. | std. dev. | Max. del | Std. dev. |
| MPEG2 | 1.8 | 0.38 | 0.163 | 2.05 | 0.38 | 0.2 | 12% | 19% |
| Decryption | 1.22 | 0.39 | 0.15 | 1.3 | 0.4 | 0.16 | 6% | 6% |
| Encryption | 1.47 | 0.45 | 0.19 | 1.47 | 0.46 | 0.21 | - | 10% |

### 4.3 Fairness Measurements

We measured the fairness under a highly congested scenario where the packet generation rates for all the flows were adjusted such that all flows at the scheduler were backlogged for the entire simulation period. We found that, with CBCS, the total allocation of resources for each flow remained more or less constant at any time for all the flows. Fig. 2a shows a snapshot of the actual CPU allocations and the corresponding demanded bandwidth allocations per flow measured at the $300\text{th}$ sec using CBCS. On the other hand, Fig. 3a shows the actual CPU and bandwidth allocations per flow measured at the $300\text{th}$ secs using CBCS. Since the CBCS scheduler considers both the CPU and bandwidth resource requirement while serving packets from a flow, this figure shows that the demanded BW requirements for each flow is equal to actual BW allocation for the same flow. Also, this figure shows that CBCS maintained perfect fairness by keeping the total resource allocations for each flow more or less constant. The scheduler allocated more CPU for flows 1 to 10 and more bandwidth for flows 11 to 30. The total CPU utilization was 100 percent, the demanded bandwidth utilization was 100 percent, and the actual bandwidth utilization was 99 percent using CBCS. Fig. 2b shows the actual CPU allocations and corresponding demanded bandwidth per flow measured at the $300\text{th}$ second during the simulation using the DRR schedulers. Fig. 3b shows the actual CPU and bandwidth allocations per flow measured at the $300\text{th}$ sec using the separate DRR schedulers. These results demonstrate that maintaining fairness in one allocation of one resource does not necessarily guarantee fairness in the allocation of other resources. As shown in Fig. 2b, although the CPU scheduler maintained perfect fairness in allocating CPU, it served a fewer number of packets from flows 1 to 10, which have high CPU requirements and more packets than flows 11 to 30, which have lower CPU and higher bandwidth requirements. As a result, the demanded bandwidth for flows 11 to 30 were very high compared to that of flows 1 to 10, as shown in Fig. 3b. Although the actual CPU utilization was 100 percent, the demanded bandwidth requirements for all the flows combined was 245 percent. This high bandwidth demand resulted in significant packet loss for flows 21 to 30 in the bandwidth scheduler. The actual bandwidth utilization was also 99.97 percent. As shown in Fig. 3b, since the demanded bandwidth for flows 1 to 10 was less than their fair share, the bandwidth scheduler equally distributed the available bandwidth among flows 11 to 30.

TABLE 3
Reduction in Packet Loss Using CBCS

| Flow Number | *Packet Entering the Programmable Node* | *Packet Processed at the Node* | | *Packet Received at Sink* | |
|---|---|---|---|---|---|
| | | CBCS | DRR | CBCS | DRR |
| 1 | 4285 | 1189 | 4033 | 1185 | 1061 |
| 2 | 4285 | 1193 | 4031 | 1190 | 1037 |
| 3 | 4285 | 1172 | 4062 | 1167 | 1025 |
| 4 | 4285 | 1189 | 4045 | 1185 | 1048 |
| 5 | 4285 | 1181 | 4041 | 1177 | 1056 |

### 4.4 Evaluation of Instantaneous Fairness Using the Gini Index

All the fairness measures such as RFB and the worst-case fair index [2] only measure the worst-case behavior of any scheduler. They do not provide insights into the instantaneous fairness that could be achieved by a scheduler [24]. Consider a situation where two schedulers have the same worst-case fairness bound, but one scheduler rarely operates at the upper bound, whereas the other consistently achieves the worst-case difference in the resource allocation. The worst-case bound cannot help differentiate between the two despite the fact that the latter scheduler is more fair. Hence, there is a need for a metric that measures the fairness achieved by a scheduler at any given instant of time. A novel metric called the *Gini Index*, originally proposed in the field of economics [25], was adopted in [24] to measure the instantaneous fairness of scheduling algorithms. We present a brief overview of this metric. Readers are referred to [24] for a detailed description. The computation of the Gini index is described formally as follows:

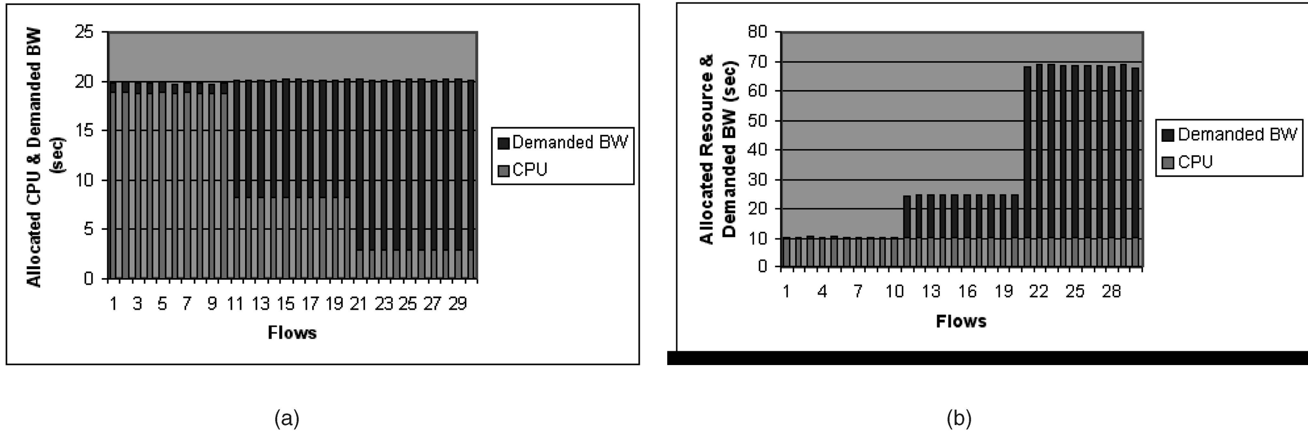
**Definition 4.** *Let $U(t)$ represent the set of the session utilities of the flows at time instant $t$ when served by a real scheduler and let $G(t)$ denote a similar set which is obtained when the flows are served with the ideal GPS scheduler. Let $u_{c1}, u_{c2}, \ldots, u_{ck}$ be the elements of the set $U(t)$, such that $u_{c1} \leq u_{c2} \leq \ldots \leq u_{ck}$. The Lorenz Curve of the set of session utilities $U(t)$ is the function $F(i, U(t))$ given by*

$$F(i, U(t)) = \sum_{j=1}^{i} u_{cj}, \quad 0 \leq i \leq k.$$

*The Gini Index over the $k$ elements in $U(t)$ is computed as*

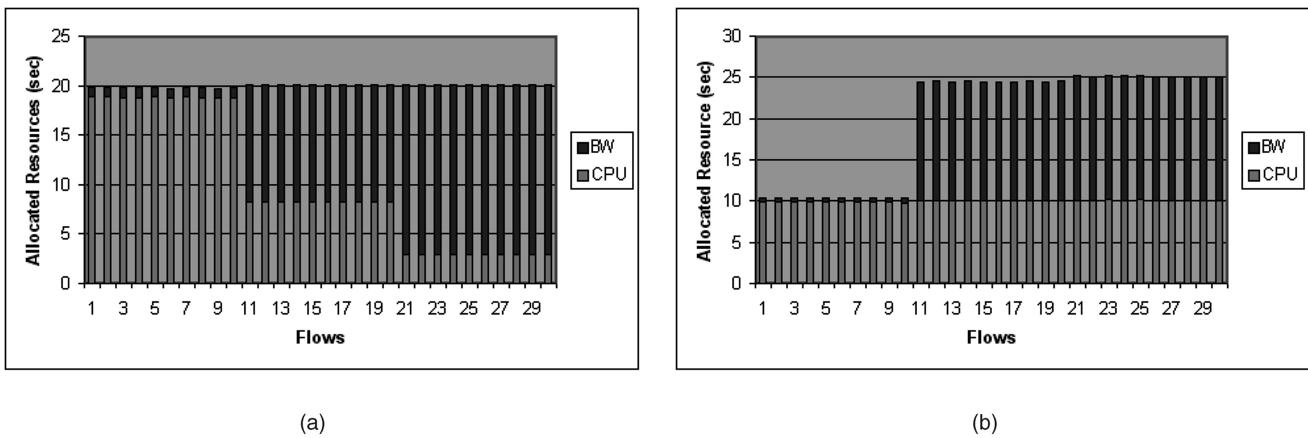
$$\sum_{i=1}^{k} |F(i; U(t)) - F(i; G(t))|. \tag{17}$$

From the definition of the virtual time function, we know that the normalized service received by each flow at time $t$




(a) (b)

Fig. 2. CPU allocation and corresponding demanded BW requirements per flow measured at the $300\text{th}$ sec using CBCS and DRR schedulers. (a) CBCS scheduler. (b) DRR scheduler.




(a) (b)

Fig. 3. Actual CPU and BW allocation at the $300\text{th}$ sec using CBCS and DRR schedulers. (a) CBCS scheduler. (b) DRR scheduler.

in the GPS system is equal to the virtual time, $V(t)$. Hence, the Gini index can be computed as

$$\sum_{i=1}^{k} |F(i;U(t)) - F(i;V(t))|. \tag{18}$$

Figs. 4 and 5 show the instantaneous fairness in the allocation of CPU and bandwidth with CBCS and separate DRR schedulers, respectively. Recall that, the lower the gini index, the better are the fairness properties of the scheduler. Notice that there are more spikes in the Gini index with DRR implying that CBCS has better fairness characteristics.

## 5 Implementation and Performance Measurements of CBCS on a Network Processor

With the rapid increase in the capacity of transmission links, the ease with which a scheduler can be implemented in real hardware systems gains paramount importance. In this section, we demonstrate how the CBCS scheduler can be easily implemented on an off-the-shelf network processor such as the Intel IXP2400 network processor [1]. We also evaluate the performance of our scheduler and compare it with an implementation consisting of separate DRR schedulers for the CPU and the link bandwidth.

### 5.1 Implementation Details

We have developed a data plane application for the IXP2400 and implemented both CBCS and also two sets of separate CPU and bandwidth schedulers, based on DRR, on the fast path processing, i.e., on the microengines. Our application consists of modules for Packet Rx, Processing, Packet Tx, Queue Manager, and the Scheduler. Also, the Ethernet layer 2 encapsulation is included in the packet-processing block. The implementation platform consists of a dual boot workstation, an IXP2400 PCI card, and Intel IXA (Internet Exchange Architecture) 3.1 SDK and framework. The IXA 3.1 framework also includes a developer workbench or Integrated Development Environment (IDE). The development workstation is a Linux workstation configured to allow the use of Windows 2000 hosted tools using VMware. VMware allows running the IXA SDK developers workbench under Microsoft Windows 2000 while running Linux 7.3 as the host operating system. The workstation has a Pentium 4 1.5-GHZ CPU and 512 MB of RAM.

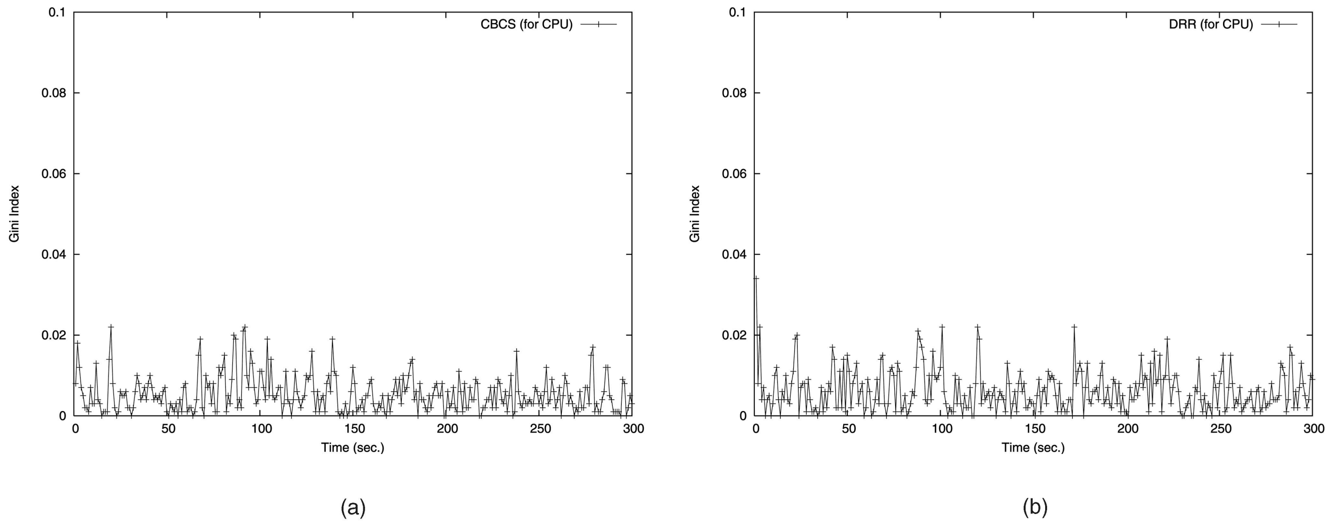



Fig. 4. Comparison of Gini indices for CPU scheduling. (a) CBCS. (b) DRR.

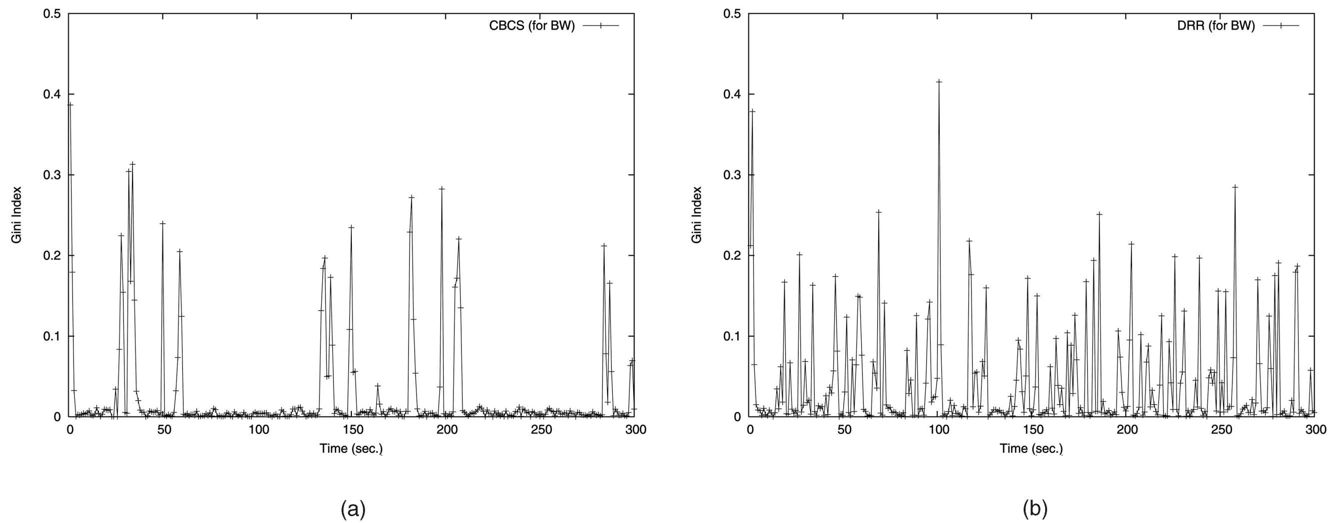



Fig. 5. Comparison of Gini indices for bandwidth scheduling. (a) CBCS. (b) DRR.

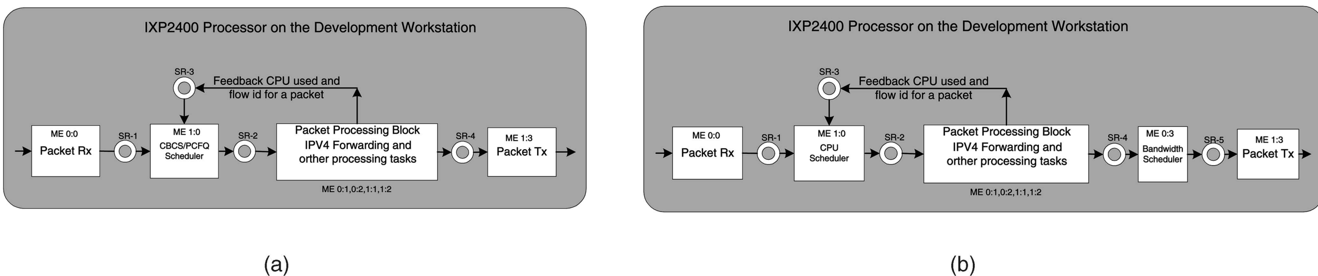



Fig. 6. Scheduler implemntation architecture. (a) CBCS scheduler. (b) Separate CPU and bandwith scheduler.

### *5.1.1 CBCS Implementation*

We have implemented the CBCS scheduler on a single microengine because it is not efficient to run the enqueue and dequeue method of the same scheduler in different microengines. The implementation architecture of the schedulers is shown in Fig. 6a. The scheduler is implemented before the packet-processing block. The packet Rx microengine receives the packets and sends an enqueue message to the scheduler via scratchpad ring 1(SR-1). The scheduler microengine continually reads the enqueue request from SR-1, estimates the CPU requirements of the packet using the SES estimations technique, and enqueues the packet info in the SRAM queue. After dequeuing a packet, the scheduler sends a message to the processor microengines via a scratchpad ring (SR-2). Packet processing code runs on four microengines and all the microengines read the processing requests from SR-2 and process

the packets. After processing the packet, the packet-processing microengines send a message specifying the CPU consumed and the flow id to the scheduler via another scratchpad ring (SR-3). After processing the packet, packet processor microengines send a transmission message to the transmitter microengine via a scratchpad ring (SR-4).

#### 5.1.2 Separate CPU and Bandwidth Schedulers Implementation

As mentioned earlier, we have also implemented a set of separate DRR schedulers for scheduling CPU and bandwidth separately on the IXP2400 processor in order to evaluate its performance in comparison to that of the CBCS scheduler. Fig. 6b shows the implementation architecture of the separate schedulers. The messages that pass through the SR-1, SR-2, and SR-3 are same as that of Fig. 6a. Here, after processing the packet, the processor microengines send an enqueue request to the bandwidth scheduler via SR-4. After dequeuing a packet, the bandwidth scheduler sends a transmission message to the Packet TX microengine via SR-5.

#### 5.1.3 Data Structures and Intermicroengines Messages

For each packet received, packet data is kept in DRAM and packet metadata (i.e., information about the packet) is kept in SRAM. The packet metadata structure has eight long word members. The IXP library provides macros and functions called dispatch loop functions to read packet metadata from SRAM and to write the metadata back into SRAM. A dispatch loop combines microblocks on a microengine and implements the data flow between them. The dispatch loop also caches commonly used variables in registers or local memory. These variables can be accessed by microblocks using the dispatch loop macros and functions. Dispatch loop functions were used to write some data like the total resource requirement for a packet into a member of the packet metadata in SRAM during a packet enqueue operation and to retrieve the data back from SRAM during a packet dequeue operation.

#### 5.1.4 CBCS Implementation Details

We have used microengine local memory for keeping CBCS scheduler variables such as Quantum, packet counts for the flows or queues, credit counter per flow, estimated CPU requirements (per packet per flow), etc. We used the local memory as it is the fastest to access and it was enough to accommodate our variables for our experiments, which consisted of 16 flows. However, SRAM can be used for allocating the variables when the number of flows is extremely high.

The CBCS scheduler is implemented using four threads: initialization thread, enqueue thread, dequeue thread, and CPU prediction thread. After initialization is completed, the initialization thread sends signals to the enqueue, dequeue, and CPU prediction threads to begin their tasks as they wait on the initialization thread's completion signal.

*Initialization Thread*. The initialization thread sets the SRAM channel CSR to indicate that packet-based enqueue and dequeue would be done, i.e., each enqueue and dequeue operation deals with a full packet. The thread also initializes SRAM queue descriptors, queue arrays, and all the scheduler variables including the quantum, credit counter for the flows, estimated CPU requirements per flow, etc. After initializing the scheduler variables, the thread terminates itself so that the microengine thread arbiter excludes this thread from its list.

*Enqueue Thread*. Fig. 7a shows a simplified flow diagram of works performed within the CBCS enqueue thread. The enqueue thread waits for the signal from the initialization thread before starting its infinite loop. In each turn, the thread calls an SRAM API (e.g., scratch_get_ring) to read an enqueue message from SR-1 and specifies a signal number as a parameter to the API call. The thread then swaps out to allow other threads to run as the SRAM read operation would take some time. After receiving the control back, the thread checks the presence of the signal, i.e., it checks whether the enqueue message read operation is completed or not. Once the enqueue message is read, it checks the validity of the enqueue message as there may not be any message in the ring.

If the thread receives an invalid message, it does a context swap and then goes for the next turn. The third LW of packet metadata contains the packet size field. So, if the enqueue message is a valid message, the thread reads the third LW of the packet metadata from the SRAM using another API (e.g., sram_read) and extracts the packet size for calculating the total resource requirement (i.e., both the CPU and bandwidth) for the packet. The CPU requirement data is taken from the global variable (per flow), which is constantly updated by the CPU prediction thread. The calculated total resource requirement is used by the dequeue thread for scheduling purposes and, therefore, it needs to be stored. We decided to use the seventh LW of the packet metadata to store this scheduler data.

The enqueue thread calls an SRAM API (e.g., sram_ write) to write back the resource requirement data to SRAM and specifies a signal number. While the write operation is in progress, the thread calls another API to enqueue the packet info the SRAM queue corresponding to the flow-id. The enqueuing is processed using the packetNext pointer. The thread increments the packet count for the queue and waits for the SRAM write operation to be completed. The thread then does a context swap and goes for the next round.

*RR Calculations*. We calculate the total resource requirement (RR) for the incoming packets in nano seconds (ns) using the following equation:

$$\begin{aligned} RR &= \text{CPU Cost of the packet (ns)} \\ &\quad + \text{Transmission cost of the packet (ns)} \\ &= \text{CPU cost (ns) per CPU Cycle} * \text{Estimated CPU Cycles} \\ &\quad + \text{Transmission cost per byte (ns)} * \text{Packet size in Bytes.} \end{aligned}$$

Each microengine has a clock frequency of 600 MHZ, i.e., 600 millions cycles per sec. Therefore, the CPU cost per $\text{CPU Cycle} = \frac{5}{3} ns$. For a 100 Mbits network interface, the transmission cost per byte would be 80 ns. Since the microengines do not support the floating-point calculations, the CPU cost calculation for a packet is approximated, where the calculation error is less than or equal to $\frac{2}{3}$ ns. This calculation error or approximation is quite acceptable as it is tiny compared to the value of $RR$ and it happens for some of the packets for all flows.

*Dequeue Thread*. Fig. 7b shows the simplified flow diagram of the activities performed within the CBCS

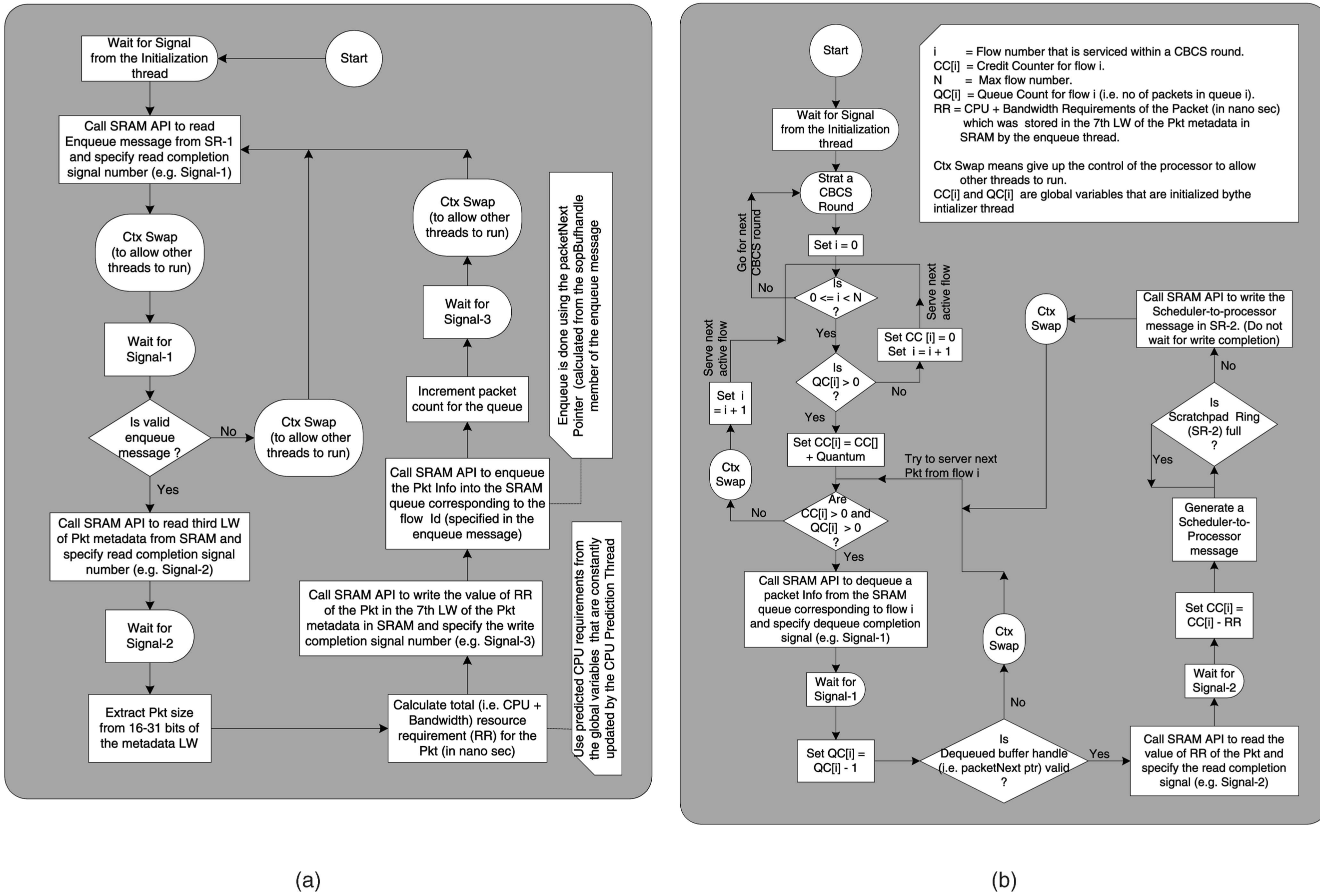



Fig. 7. Flow diagram of the CBCS enqueue and dequeue thread. (a) CBCS enqueue thread. (b) CBCS dequeue thread.

dequeue thread. As shown, the dequeue thread waits for a signal from the initialization thread before starting its infinite loop. In each CBCS round, the algorithm serves all the active flows. For each flow $i$, the algorithm checks whether the Queue Count, i.e., $QC[i]$, which is stored in global variables, is positive. If $QC[i]$ is positive, it adds the quantum to the value of the Credit Counter of the flow $i$ (i.e., $CC[i]$); otherwise, it resets the $CC[i]$ to 0 and tries to serve the next active flow.

While serving flow $i$ within each CBCS round, the algorithm checks whether both the $CC[i]$ and the $QC[i]$ are positive or not. If either of them is 0 or negative, the algorithm does a context swap so that other threads get a chance to run and then tries to serve the next active flow. Otherwise, the algorithm calls an SRAM API (e.g., sram_dequeue) to dequeue the packet info from the SRAM queue corresponding to flow $i$ and it waits for the dequeue completion signal. After the dequeue, it decrements the queue count for flow $i$ and then it checks the validity of the dequeued buffer handle (i.e., the packetNext ptr as enqueued in the enqueue operation). If the buffer handle is invalid, it does a context swap and then tries to serve the next packet from the same flow $i$.

For a valid dequeue of a packet, the code calls another SRAM API to read the resource requirement, $RR$, which is the CPU requirement plus bandwidth requirement in nano seconds and waits for the read operation to complete. On completion of the SRAM read, the system signals the thread and the code then decrements the $CC[i]$ by the value of RR. The thread then generates a scheduler-to-processor message and enqueues the message to the scratchpad ring 2 (SR-2). However, before enqueuing, it checks the fullness of the ring using IXP API and waits if the ring is full. After sending the message to the processor, the thread swaps out and tries to serve the next packet from the same flow $i$.

*CPU Prediction Thread*. This thread waits for the signal from the initialization thread before it starts its infinite loop. In each turn, the thread calls an SRAM API to read the processor-to-scheduler message from scratchpad ring 3 (SR-3) and specifies a signal number to wait on and then swaps out so that other threads can work while it is waiting for the read to complete. Upon receiving a valid message, it updates the estimated CPU requirement of the specified flow using SES estimation as discussed in Section 2. The estimated per packet CPU requirements are kept in global variables.

### 5.2 Experiments and Results

The experiments were performed by running the code on the IXA workbench's cycle accurate transactor. We could only evaluate the delay characteristics for the CBCS and separate DRR schedulers because of the limitations of the workbench simulator, which only provides the input and output port logging options.

In our experiments, we used 16 flows with varying packet sizes and different CPU requirements. Table 4 shows

TABLE 4
Experimental Setup

| Flow Number | CPU Req. Category | Bandwidth Req. Category | CPU Req. (Cycles) | Packet Size (Bytes) |
|---|---|---|---|---|
| 1, 5, 9, 13 | High | Low | 2400 - 3600 | 42 - 48 |
| 2, 6, 10, 14 | Low | High | 78 - 134 | 120 - 127 |
| 7, 16 | Medium | Medium | 1200 - 1800 | 80 - 88 |
| 3, 8, 12 | Low | Low | 78 - 134 | 42 - 48 |
| 4, 11, 15 | High | High | 2400 - 3600 | 120 - 127 |

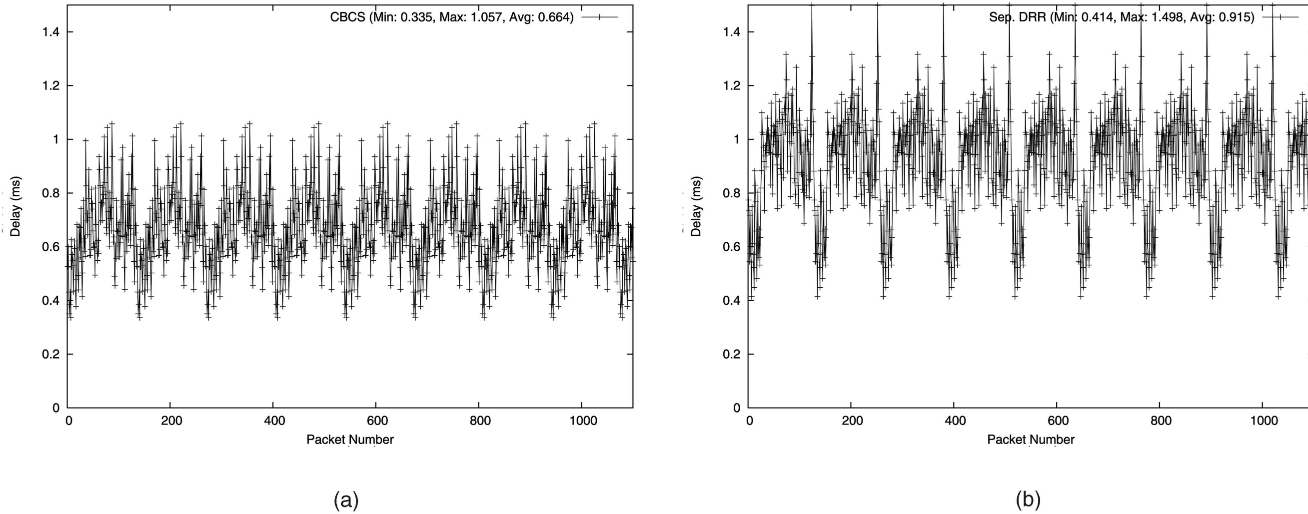



Fig. 8. Delay for flows with high CPU and high BW requirements. (a) CBCS. (b) DRR.

the CPU requirements and packet sizes for each individual flow. We broadly classify the flows into five groups depending on their CPU and bandwidth requirements. We were unable to reproduce the same set of experiments as in our simulation study discussed in Section 4 due to the difficulty of porting our MPEG2 encoding and RC2 encryption and decryption code to the IXP processor. Further, the inability of the IXP2400 to support floating point calculations restricted our processing tasks to simple examples such as IPv4 forwarding, which were provided with the IXP SDK.

For system settings, the default settings of the workbench simulator were used. Each microengine has a clock frequency of 600 MHz. The SRAM clock frequency was set to 200 MHz and the DRAM frequency was set to 150 MHz. The PLL output frequency used was 1,200 MHz. The receive and transmission rates on the media interfaces were set to 50 Mbps. We created the 16 data stream files containing Ethernet frames and used the workbench simulator's network traffic assignment functionality to inject the data frames. The SES coefficient $\alpha$ was set to 0.4.

The packet logs obtained from the IXP processor were analyzed to determine the packet delays. Figs. 8 and 9 show the delays measured for one particular flow from the two groups (flow with high CPU and high BW requirement and low CPU and high BW requirement) for the CBCS and separate DRR schedulers. We have observed that the results for the other flows within the same group and also for other groups are quite similar to the representative flows used in the comparisons. Due to space limitation, we could not show all the delay graphs. The maximum and average delays and the standard deviation for these flows are shown in Table 5. Delay results show that CBCS achieved superior delay guarantees compared to DRR (when used individually for CPU and bandwidth scheduling) for all the flows.

It should be noted that the performance improvement achieved by CBCS in our IXP experiments is slightly lower than that observed in the simulations discussed in Section 4. However, direct comparisons between these two sets of results is not possible due to the vastly different nature of the underlying simulation platforms: The ns-2 simulator is a single-threaded event-driven system, whereas the IXP workbench transactor is a cycle accurate simulator and the code compiled for the transactor can be run on the IXP board itself. Further, the IXP processor is unable to execute floating point operations, thus limiting the processing operations that can be requested by the flows. This also necessitates using different parameters in these two experiments, notably, the flow characteristics. However, the IXP experiments do indicate that CBCS does demonstrate noticeable performance improvements over separate DRR schedulers.

## 6 Conclusion

In this paper, we have presented the design, implementation, and evaluation of a novel composite scheduler called *Composite Bandwidth and CPU Scheduler (CBCS)* for dynamical scheduling of multiple resources. Analytical evaluations of the work complexity and fairness bounds for the CBCS scheduler were presented. We have compared our scheduler

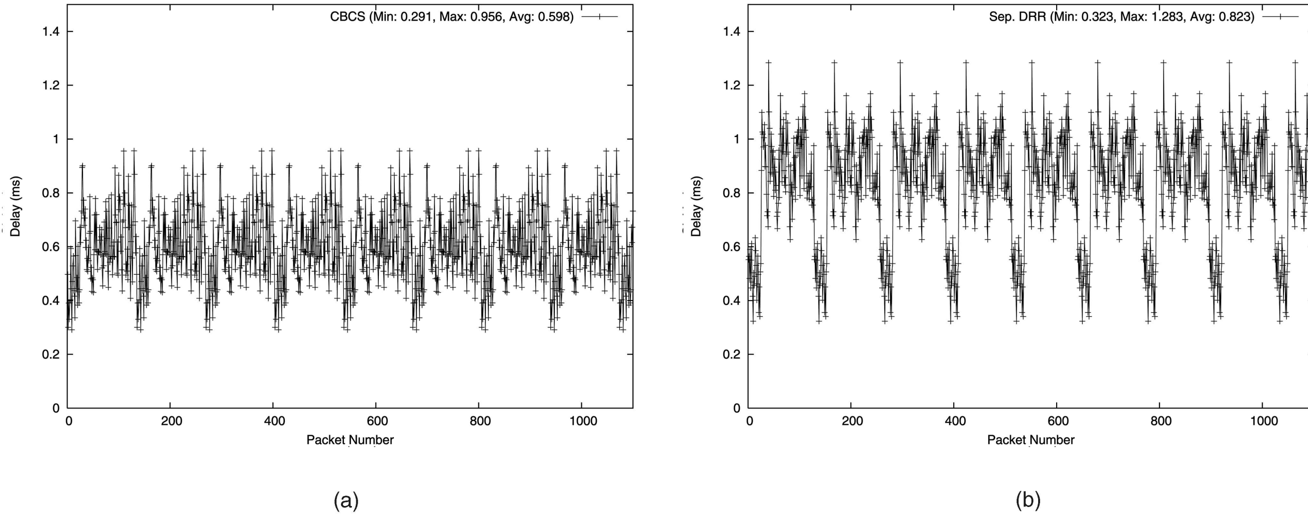



Fig. 9. Delay for flows with low CPU and high BW requirements. (a) CBCS. (b) DRR.

TABLE 5
Statistics for the Delay Characteristics

| Flow Groups | CBCS Scheduler Statistics (in sec) | | | Separate DRR Scheduler Statistics (in sec) | | | Performance Improvement using CBCS | | |
|---|---|---|---|---|---|---|---|---|---|
| | Max Delay | Avg Delay | Std Dev | Max Delay | Avg Delay | Std Dev | Max Delay | Avg Delay | Std Dev |
| High CPU, High BW | 1.06 | 0.66 | 0.15 | 1.5 | 0.91 | 0.19 | 29% | 28% | 25% |
| High CPU, Low BW | 1.11 | 0.71 | 0.14 | 1.3 | 0.88 | 0.19 | 15% | 21% | 22% |
| Low CPU, High BW | 0.96 | 0.60 | 0.14 | 1.28 | 0.82 | 0.21 | 26% | 28% | 35% |
| Low CPU, Low BW | 0.99 | 0.59 | 0.14 | 01.09 | 0.77 | 0.19 | 9% | 25% | 26% |
| Med. CPU, Med. BW | 1.11 | 0.72 | 0.14 | 1.31 | 0.89 | 0.21 | 15% | 19% | 35% |

with a system consisting of separate schedulers for allocating the link and processor resource, which is typical of most routers today. Our analytical results show that CBCS is a low complexity scheduler ($O(1)$) which has better fairness and performance characteristics as compared to an implementation consisting of separate schedulers of similar complexity. In addition, we have also presented simulation results which corroborate the conclusions of our analysis. The performance results strongly support the benefits of our composite scheduler to better support computing along the forwarding path without sacrificing efficiency.

With the rapid growth in link bandwidth, the duration of time that is available to a router for making a scheduling decision is diminishing rapidly. Hence, it is imperative that a scheduling algorithm can be easily implementable in real hardware systems. In this paper, we developed a real-world implementation of the CBCS scheduler using an off-the-shelf network processor such as the Intel IXP 2400. We also presented empirical evaluations to highlight the improved delay characteristics of CBCS in comparison with an implementation consisting of two separate DRR schedulers.

Even though we have focused on the joint allocation of the processing and bandwidth resources, our algorithm can be readily adapted for the joint allocation of a combination of different heterogeneous resources such as bandwidth and battery power in mobile ad hoc networks, memory, and processor cycles in routers.

## Acknowledgments

This research was supported by the Smart Internet Technology Co-operative Research Centre (SITCRC), Australia. The authors would also like to thank Intel Corp. for providing the IXP2400 processor boards as part of the IXA Education Program.

## References

[1] "Intel IXP2400 Network Processor Overview," white paper, http://www.intel.com/design/network/products/npfamily/ixp2400.htm, 2007.

[2] 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, no. 3, pp. 344-357, June 1993.

[3] D.C. Stephens, J.C.R. Bennett, and H. Zhang, "Implementing Scheduling Algorithms in High-Speed Networks," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1145-1158, June 1999.

[4] P. Goyal, H.M. Vin, and H. Cheng, "Start-Time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *IEEE/ACM Trans. Networking,* vol. 5, no. 5, pp. 690-704, Oct. 1997.

[5] P. Pappu and T. Wolf, "Scheduling Processing Resources in Programmable Routers," *Proc. IEEE INFOCOM '02,* June 2002.

[6] C.W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems,* pp. 90-99, May 1994.

[7] K. Lakshman, R. Yavatkar, and R. Finkel, "Integrated CPU and Network I/O QoS Management in an Endsystem," *Proc. Fifth Int'l Workshop Quality of Service (IWQOS '97),* pp. 167-178, 1997.

[8] V. Galtier, K. Mills, and Y. Carlinet, *Predicting and Controlling Resource Usage in a Heterogeneous Active Network.* Nat'l Inst. of Standards, 2001.

[9] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a Robust Software-Based Router Using Network Processors," *Proc. 18th ACM Symp. Operating Systems Principles,* Dec. 2001.

[10] J. Yao, J. Guo, L. Bhuyan, and Z. Xu, "Scheduling Real-Time Multimedia Tasks in Network Processors," *Proc. IEEE GLOBECOM,* Nov. 2004.

[11] W. Shi, X. Zhuang, I. Paul, and K. Schwan, "Efficient Implementation of Packet Scheduling Algorithm on High-Speed Programmable Network Processors," *Proc. IEEE Int'l Conf. Management of Multimedia Networks and Services,* pp. 184-197, Oct. 2002.

[12] F. De Bernardinis, "A QoS Internet Protocol Scheduler on the IXP1200 Network Platform," *Proc. Third IEEE Int'l Workshop System-on-Chip for Real-Time Applications (IWSOC '03),* June 2003.

[13] A. Demers, S. Keshav, and S. Shenker, "Design and Analysis of a Fair Queuing Algorithm," *Proc. ACM SIGCOMM,* pp. 1-12, Sept. 1989.

[14] S.J. Golestani, "A Self-Clocked Fair Queuing Scheme for Broadband Applications," *Proc. IEEE INFOCOM '94,* pp. 636-646, June 1994.

[15] M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin," *IEEE/ACM Trans. Networking,* vol. 4, no. 3, pp. 375-385, June 1996.

[16] S. Kanhere, H. Sethu, and A. Parekh, "Fair and Efficient Packet Scheduling Using Elastic Round Robin," *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 3, pp. 326-336, Mar. 2002.

[17] V. Ramachandran, R. Pandey, and S.-H. Chan, "Fair Resource Allocation in Active Networks," *Proc. IEEE Int'l Conf. Computer Comm. and Networks (ICCCN '00),* pp. 468-475, Oct. 2000.

[18] F. Sabrina and S. Jha, "Fair Resource Allocation in Programmable and Active Networks Using a Composite Scheduler," *Proc. Seventh Int'l Symp. Digital Signal Processing and Comm. Systems (DSPCS '03),* 2003.

[19] F. Sabrina and S. Jha, "Fair Queuing in Active and Programmable Networks," *Advanced Wired and Wireless Networks Series: Multimedia Systems and Applications,* vol. 26, 2004.

[20] F. Sabrina and S. Jha, "Scheduling Resources in Programmable and Active Networks Based on Adaptive Estimations," *Proc. 28th Ann. IEEE Conf. Local Computer Networks (LCN),* pp. 2-11, Oct. 2003.

[21] S.K. Jha, P.A. Wright, and M. Fry, "Playout Management of Interactive Video—An Adaptive Approach," *Proc. Fifth Int'l Workshop Quality of Service (IWQOS '97),* pp. 145-156, 1997.

[22] S. Kanhere and H. Sethu, "Low-Latency Guaranteed-Rate Scheduling Using Elastic Round Robin," *Computer Comm.,* vol. 25, no. 14, pp. 1315-1322, Sept. 2002.

[23] *The Network Simulator—ns-2,* http://www.isi.edu/nsnam/ns/, 2007.

[24] H. Shi and H. Sethu, "An Evaluation of Timestamp Based Packet Schedulers Using a Novel Measure of Instantaneous Fairness," *Proc. IEEE Int'l Conf. Performance, Computing, and Comm.,* Apr. 2003.

[25] F.A. Cowell, *Measuring Inequalities: Techniques for Social Sciences.* John Wiley and Sons, 1977.

**Fariza Sabrina** received the BSc degree in electrical and electronics engineering from BUET, Dhaka, Bangladesh, in 1997 and the MER degree in electrical and information engineering from the University of Sydney, Australia, in 2000. She received the PhD degree in computer science and engineering from the University of New South Wales, Sydney, Australia, in 2005. Since 2005, she has been working as a research fellow in the Networking Technologies Lab at the Commomenwealth Scientific and Industrial Research Organisation (CSIRO), Sydney, Australia. Her research interests include quality-of-service, resource, and traffic management in computer networks, embedded and real-time systems, programmable and multimedia networks, and network security. She is a member of the IEEE, the IEEE Computer Society, and the ACM.

**Salil S. Kanhere** received the BE degree in electrical engineering from the University of Bombay, Bombay, India, in 1998 and the MS and PhD degrees in electrical engineering from Drexel University, Philadelphia, in 2002 and 2003, respectively. Since 2004, he has been a lecturer with the School of Computer Science and Engineering at the University of New South Wales, Sydney, Australia. His current research interests include wireless mesh networks, sensor networks, vehicular ad hoc networks, network security, and quality-of-service in computer networks. He is a member of the IEEE and the ACM.

**Sanjay K. Jha** received the PhD degree from the University of Technology, Sydney, Australia. He is a professor in the School of Computer Science and Engineering at the University of New South Wales. His research activities cover a wide range of topics in networking including wireless sensor networks, ad hoc/community wireless networks, resilience/quality of service (QoS) in IP networks, and active/programmable networks. He has published extensively in high quality journals and conferences. He is the principal author of the book *Engineering Internet QoS* and a coeditor of the book *Wireless Sensor Networks: A Systems Perspective*. Dr. Jha is a member-at-large of the Technical Committee on Computer Communication (TCCC) and a member of the IEEE. He is also a member of the editorial advisory board of the *Wiley/ACM International Journal of Network Management*. He has served on program committees of several conferences. He has also served as the technical program committee chair of the IEEE Local Computer Networks (LCN '04) and ATNAC '04 conferences and as the general chair of the Ements-II Workshop.



▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.