Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Communications 28 (2005) 676–687

computer communications

www.elsevier.com/locate/comcom

Processing resource scheduling in programmable networks

F. Sabrina[a,*], C.D. Nguyen[b], S. Jha[a], D. Platt[b], F. Safaei[b]

[a]*School of Computer Science and Engineering, UNSW, Sydney, Australia*
[b]*Telecommunications and Information Technology Research Institute, UOW, Wollongong, Australia*

Received 5 August 2004; accepted 5 August 2004
Available online 11 September 2004

Abstract

Programmable network paradigm allows the execution of active applications in routers or switches to provide more flexibility to traditional networks, and richer services for users. In this paper, we discuss issues in designing resource schedulers for processing engines in programmable networks. One of the key problems in programmable networks is the inability to determine execution times of packets from information in headers for scheduling which is in contrast to using packet length in transport resources scheduling. Therefore, this paper focuses on developing CPU scheduling algorithms that could schedule CPU resource adaptively, fairly, and efficiently among all the competing flows. In this paper, we present two scheduling algorithms that could resolve the problem of prior determination of CPU requirement of the data packet. One of the proposed packet scheduling algorithm is called start time weighted fair queueing that does not require packet processing times in advance and the other one is called prediction based fair queueing which uses a prediction algorithm to estimate CPU requirements of packet and it then schedules the packets according to that. The effectiveness of these algorithms in achieving fairness and providing delay guarantee is shown through analysis and simulation work.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Start time weighted fair queueing; Quality of service; Prediction based fair queueing

1. Introduction

Active or programmable networks [1,8] have been recognized as a future solution to provide more flexibility to the current internet. Programmable networks provide custom processing on packets at processing engines in order to enable fast deployment of new protocols and provide innovative services within the network to cater for various needs of end users. While many works in the literature have been looking at potential applications and advantages of programmable networks, there is little work considering larger scale deployment of programmable networks, especially, quality of service (QoS) support for network processing applications. Our work aims to design an efficient processing resource management at programmable routers.

Current programmable node architectures [1,8] enforce isolation of packet processing between flows or threads for security and accounting purposes. However, QoS scheduling and admission control are not specified. The work in [5] discusses the problem of scheduling computation resources in a software-based router but relies on the assumption that processing times of packets are pre determined. Galtier et al. [2] shows that the prediction of packet execution times in an experimental testbed can have large mean errors. The scheme seems too complicated to be implemented with programmable routers. They also point out that it is necessary to calibrate processing requirements of packets in different routers which have different hardware and software properties. The work in [4] shows that the processing load of some networking applications are predictable and correlated to the packet sizes. The observed correlation is then used for specifying processing resource reservations and scheduling based on estimation of each packet processing times. However, in general purpose processing, it is hard to determine the processing time of an arbitrary piece of code contained in a packet.

* Corresponding author.
E-mail addresses: farizas@cse.unsw.edu.au (F. Sabrina), dcn01@uow.edu.au (C.D. Nguyen), sjha@cse.unsw.edu.au (S. Jha), don_platt@uow.edu.au (D. Platt), farzad@uow.edu.au (F. Safaei).

In this paper, we present two novel packet scheduling algorithms called start time weighted fair queueing (SWFQ) and prediction based fair queueing (PBFQ), and show that they offer good fairness and delay properties. The rest of this paper is organized as follows: Section 2 presents some background in packet fair queueing (PFQ) disciplines. Sections 3 and 4 describe our proposed scheduling algorithms, and an analysis of the algorithm. Section 5 presents the simulation results. We draw conclusions in Section 6.

2. Design of processing resources scheduling

2.1. Background in packet scheduling disciplines

Generalized processor sharing (GPS) [3] is an ideal scheduling discipline based on a fluid flow model in which the traffic is infinitely divisible and each session is serviced simultaneously according to its weight. In packet-switched networks, however, the minimum service unit is a packet and only one traffic stream can receive service at a time. Therefore, most PFQ algorithms [12] are based on approximating GPS. Packets from different sessions are stamped with virtual finishing times and selected to schedule based on increasing order of virtual finishing times. Computing virtual finishing times requires the sizes of packets from packet headers. Most PFQ algorithms have the same way of updating virtual finishing times of packets based on the system virtual time, denoted as $V(t)$, as follows.

$$F_i^k = \max\{F_i^{k-1}, V(a_i^k)\} + \frac{L_i^k}{\phi_i} \tag{1}$$

where ϕ_i is the weight of session i and F_i^k, a_i^k, and L_i^k are the virtual finishing time, arrival time, and length of packet k in session i. The virtual finishing time F_i^k represents normalized amount of service, with respect to its share, session i has received right after packet k is served. When a PFQ algorithm schedules packets in increasing order of virtual finishing times, it aims to equalize the normalized amount of services of all backlogged sessions. The role of the system virtual time is to compute virtual finishing times of packets arriving at unbacklogged sessions in order to equalize the normalized services of these sessions with current backlogged sessions. For example, weighted fair queueing (WFQ) [3,9] defines the system virtual time as $V(t+\tau) = V(t) + (\tau/\sum_{i\in B(t)} \phi_i)$, where $B(t)$ is the set of current backlogged sessions at time t. A flow is backlogged during the time interval $(t2-t1)$ if the queue for the flow is never empty during the interval. It is noted that the slope of the system virtual time is inverse proportional to the number of backlogged sessions, therefore, it allows sessions to receive more service when the number of backlogged session decreases, and vice versa.

One of the existing PFQ algorithms in the literature that can also be used for CPU scheduling in programmable networks is start time fair queueing (SFQ) [6]. In SFQ, the system virtual time is defined as the start time of the current packet in service and is updated each time a packet begins service. Therefore, the system virtual time will stop advancing for a period of time if a burst of packets arrives at empty queues for a short interval, resulting in larger delays to packets from already backlogged sessions. This behavior is shown in the analysis carried out by Bennett and Zhang in [12].

2.2. Programmable networks applications

Applications that can process packets on a programmable node can be classified into two categories, header processing applications and payload processing applications. Header processing applications only perform 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. Examples of header processing applications include IP forwarding, transport layer classification, and QoS routing. In contrast, payload processing applications perform 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 [4]. Examples of payload processing applications are IPSec encryption, packet compression and packet content transcoding (e.g. image format transcoding), aggregation of sensor data etc.

2.3. Determining processing requirements

In [4] researchers have measured the processing times for a header processing application (IP forwarding) and three payload processing applications (cast encryption, adaptive Huffman coding and Reed-Solomon forward error correction) for processing packets of varying lengths and have identified that the execution times for payload processing applications strongly correlated to packet sizes and that for header processing applications does not depend on packet size. But they did not consider the effect of network load and operating system scheduling in their work. In this work we have performed some experiments to investigate the impact of processing load and operating system scheduling of a node on the packet processing times. We measured processing times for three payload processing applications (e.g. MPEG2 encoder, RC2 encryption, RC2 decryption) for processing data blocks of fixed sizes. The experiments were performed on a machine having a Pentium 4, 1.8 GHz CPU and running the Linux (RedHat 7.2) operating system.

We found that the processing times significantly varied in different number of executions for all the applications even though the data size remained the same for all execution. As a sample result, Fig. 1 shows the processing times consumed by an MPEG2 encoder code for 1000 repetitive executions on a MPEG2 data block having a size of 24,576 bytes. The processing times varied between 20

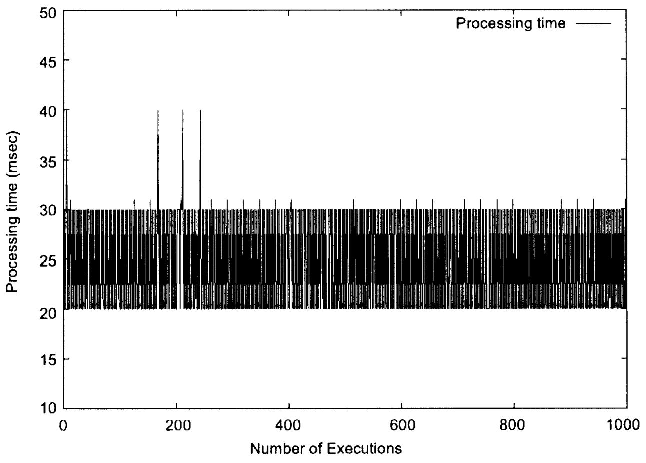


Fig. 1. Variation in actual processing time of MPEG2 data blocks of fixed length for different executions.

and 40 ms. The results indicate that processing times as measured through off-line experimentations cannot be used reliably to estimate processing times for new incoming packets referencing the same application for a scheduler that schedules packets based on estimated finish time of the packets. Therefore, designing a processor scheduler based on the estimated finish times of packets would require an online estimation process that can accommodate variation in packet size and the effect of processing load and operating system scheduling. Alternatively, we can design a processor scheduler that does rely on estimated finish times of the packets rather it relies on start time of the packet.

3. Start time weighted fair queuing (SWFQ)

From Eq. (1), if we define start time $S_i^k = \max\{F_i^{k-1}, V(a_i^k)\}$, it can represent the amount of service, normalized to its service share, session i has received before packet k is served. If a server chooses to schedule packets in increasing order of start times, it should be able to provide equivalent performance to scheduling using finishing times. Therefore, we design a PFQ algorithm based on start times since it would not require packet lengths in advance. For scheduling based on finishing times or start times, the system virtual time is important to compute finishing or start times of packets arriving at unbacklogged sessions. We present a general methodology to design PFQ algorithms based on start times as follows.

Consider a model of a processor scheduler serving N sessions which have packets with different execution time requirements. Due to unknown processing requirements of packets, only one pair of S_i and F_i is maintained for each session queue i. When packet p_i^k arrives at the head of session i queue, the start time is computed as follows.

$$\begin{cases} S_i = \max\{F_i, V(a_i^k)\} & \text{if session } i \text{ queue is empty} \\ S_i = F_i & \text{otherwise} \end{cases} \tag{2}$$

The finishing time F_i is updated only when this packet, which has processing cost P_i^k, has been processed.

$$F_i = S_i + \frac{P_i^k}{\phi_i} \tag{3}$$

Note that, this finishing time is then used to compute the start time of the next packet in session i as shown in Eq. (2). In addition, in order to avoid malicious packets that can overuse CPU resources, we define $P_i^{\max}$ as the maximum allowed processing size of packets in session i.

Using this model, we propose a PFQ algorithm called SWFQ. SWFQ uses the same system virtual time used in WFQ, but schedules packets in increasing order of start times. SWFQ is a generalization of a scheme called fair queueing based on starting time which is introduced in [13] and is based on uniform processor sharing. Fig. 2 shows how WFQ and SWFQ compute finishing times and start times to enable fair service ordering. It is noted that SWFQ only computes a start time at the head of a flow queue, therefore, it only uses packet processing times when packets have been processed. Fig. 2 also shows the worst case that, due to unknown processing time, SWFQ may misorder the first packet of a busy session compared to the service finishing order in GPS (the order of the first three packet should be 4-5-20 instead of 20-5-4). Therefore, SWFQ has different properties compared to WFQ that we will analyze in the next session.

The differences between SWFQ and SFQ are demonstrated in Fig. 2b, which shows the arrival sequences and service orders of packets in six flows, each reserving the same rates ($\phi_i = 1/6$). The server capacity is 1 unit/s; the costs of packets p_1^1 and p_6^1 are 10 units; the costs of other packets are 1 unit. At time $t=0$, p_1^1 arrives, followed by p_2^1 just after $t=0$. Packets p_3^1, p_4^1, p_5^1, and p_6^1 arrive at time $t=1$, 2, 3, and just before $t=10$, respectively. In SWFQ, during the processing time interval of p_1^1, the system virtual times increase from $V(0)=0$ to $V(10)=14.9$, and the start times of p_2^1, p_3^1, p_4^1, p_2^2, p_5^1, p_3^2, p_4^2, p_5^2, and p_6^1 are 0, 3, 5, 6, 6.5, 9, 11, 12.5, and 14.9, respectively. Consequently, the service order of SWFQ is similar to WFQ, in which the service of each session finishes in the same order as if GPS was running. In SFQ, the system virtual time remains unchanged when p_1^1 is in service. Therefore, p_2^1, p_3^1, p_4^1, p_5^1, and p_6^1 have the same start times equal to 0 and are selected arbitrarily. As shown in Fig. 2b, in the worst case, SFQ can misorder packet considerably, resulting in large delays to early arrived packets such as p_2^1. We will show later in simulations that SWFQ can improve the delay behavior of SFQ due to its more accurate system virtual time.

3.1. Analysis of SWFQ algorithm

3.1.1. Throughput analysis

Theorem 1. *For all time t and sessions i,*

$$\hat{W}_i(0,t) - W_i(0,t) \leq P^{\max} \tag{4}$$

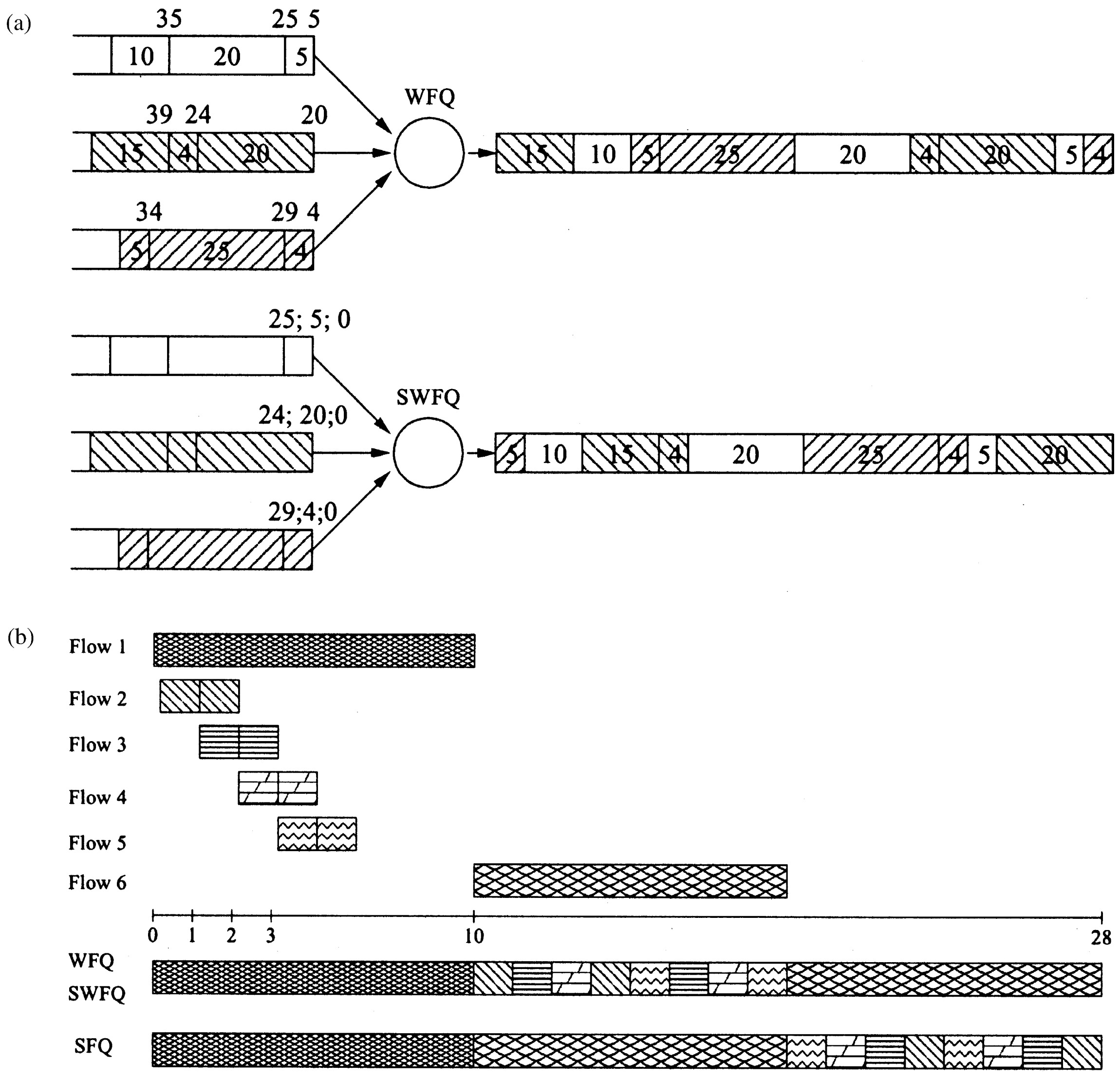


Fig. 2. Example of WFQ, SWFQ, and SFQ.

where $\hat{W}_i(0,t)$ and $W_i(0,t)$ represent work done in the SWFQ server and the GPS server in the time interval (0,t), respectively; P^{max} is the maximum processing size of a packet in the system. This theorem states maximum amount of service by which any session under SWFQ can exceed GPS. Because GPS and SWFQ are both work-conserving, they have the same busy period. Therefore, it is sufficient to prove the result for each busy period. Without loss of generality, we prove the theorem in Appendix A with the assumption that session i is constantly backlogged in interval (0,t).

Theorem 2. *For all time t and sessions i,*

$$W_i(0,t) - \hat{W}_i(0,t) \le \min\left((N-1)P^{\max}, r_i \max_{1\le n\le N}\left(\frac{P_n}{r_n}\right)\right) \quad (5)$$

where r and r_i are the server rate and the reserved rate of session i. P_n is the processing size of a packet in session n, and N is the number of sessions sharing the server. The theorem states the maximum amount of service in any session under SWFQ that can lag GPS. The proof of this theorem is given in Appendix A.

It is interesting to see that the bound on the throughput discrepancy of SWFQ and WFQ compared to GPS are the same but in opposite directions. From the result of an analysis in [14], WFQ can lead GPS by the same amount as SWFQ can lag GPS in Theorem 2. In addition, from the result of Theorem 1, SWFQ can lead GPS by the same amount that WFQ can lag GPS in [3]. Therefore, SWFQ has comparable fairness to WFQ.

3.1.2. Delay analysis

Theorem 3. *For all sessions i,*

$$L_{\mathrm{SWFQ}}(p_i^k) \le \mathrm{EAT}(p_i^k) + \sum_{n\in B(t),n\neq i} \frac{P_n^{\max}}{r} + \frac{P_i^{\max}}{r} \quad (6)$$

$L_{\mathrm{SWFQ}}(p_i^k)$ is the delay guarantee for packet p_i^k which is defined as the bound on the departure time of this packet based on its expected arrival time (EAT) [10,11]. EAT (p_i^k) is defined as

$$\mathrm{EAT}(p_i^k) = \max\left\{a(p_i^k), \mathrm{EAT}(p_i^{k-1}) + \frac{p_i^{k-1}}{r_i}\right\}$$

where $\mathrm{EAT}(p_i^0) = -\infty$. The proof is given in Appendix A. For WFQ, this delay guarantee, given in. [6] is

$$\mathrm{EAT}(p_i^k) + \frac{P_i^{\max}}{r_i} + \frac{P^{\max}}{r}$$

Note that, since the delay guarantee here is independent of the session sending rate specification, it is not the delay bound. If session rates are controlled, e.g. by a leaky bucket, the delay bound can be derived from this delay guarantee [11].

The attractive feature of PFQ using finishing times like WFQ is the ability to guarantee a delay to a session flow based on the flow's properties such as reserved rate r_i and session maximum packet processing size $P_i^{\max}$. On the other hand, the delay guarantees of SWFQ depend on the maximum processing size of packets in all backlogged sessions at the server. In some situations, SWFQ can provide smaller delay guarantees than WFQ. For example, consider session i that has larger $P_i^{\max}$ compared with other sessions and all sessions reserve the same rate. It follows that $\sum_{n\in B(t),n\neq i}(P_n^{\max}/r) < (P_i^{\max}/r_i)$, therefore, $L_{\mathrm{SWFQ}}(p_i^k) < L_{\mathrm{WFQ}}(p_i^k)$. In short, although SWFQ does not have the attractive delay guarantee feature of WFQ, it can provide predictable delay guarantees which can be acceptable in the context of processing resources scheduling.

4. Prediction based fair queuing (PBFQ)

4.1. Online estimation process

We have investigated some smoothing methods and their suitability in the context of using them within a CPU scheduler for predicting processing requirements. (Details of the investigations and comparative performance and suitability analysis of the alternatives are outside the scope of this paper.) We found that the single exponential smoothing (SES) technique can be used to estimate the execution times of the packets. SES is computationally simple and an attractive method of forecasting. Some researchers have used this method to forecast the display cycle time (which includes decompression time plus rendering time) for compressed video data packets [7]. SES uses Eq. (7) to calculate a new predicted value

$$F_{t+1} = \alpha X_t + (1-\alpha)F_t \quad \text{where } 0 \le \alpha \le 1 \tag{7}$$

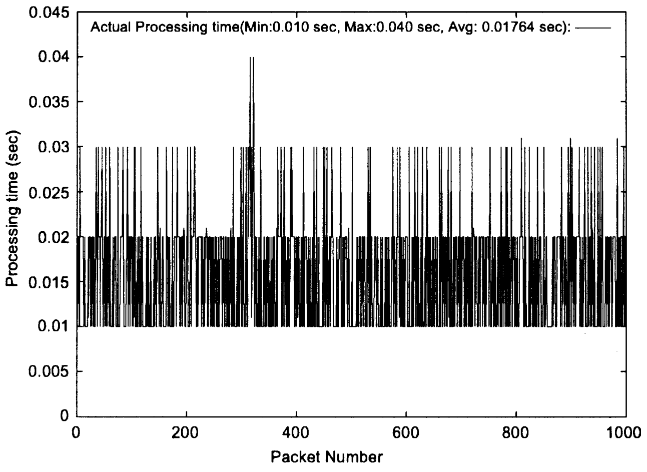


Fig. 3. Actual processing times of MPEG2 data blocks of variable lengths.

where

F_t	predicted value for the tth period
X_t	current actual value
F_{t+1}	predicted value for period $(t+1)$, i.e. next predicted value
α	parameter chosen by the user. Small value of the parameter would give more weight to the history and hence the new calculated value would not fluctuate with spikes in the data.

Figs. 3 and 4 show performance of the estimation technique on processing an MPEG2 data flow where the flow contained data blocks of varying lengths, e.g. 24,576, 13,824, 6144, and 1536 bytes. The data blocks with varying lengths were generated randomly and fed to the MPEG2 encoder code for encoding. We measured the actual processing times elapsed for 1000 data blocks and the estimated times predicted by the prediction scheme. Fig. 3 shows the actual processing times elapsed. Fig. 4 shows the times predicted by the SES technique (with the value of $\alpha = 0.5$). It may be mentioned that we performed

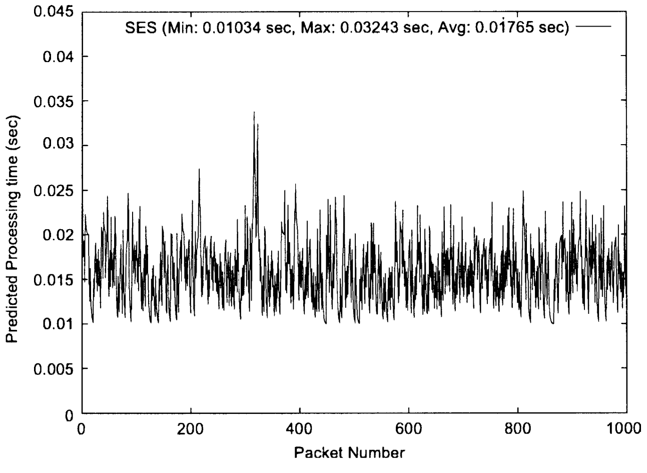


Fig. 4. Predicted processing times of MPEG2 data blocks of variable lengths using SES.

experiments using the value of α ranged from 0.2 to 0.8 and found that the predicted processing time resembles most with actual processing time when α was set to 0.5.

The figures show that the prediction technique produced estimated results that could be quite acceptable for a CPU scheduler. The patterns of the predicted values resembled the patterns of the actual processing times. It may be noted that the actual processing times varied between 10 and 40 ms with an average of 17.64 ms. Predicted times using SES varied between 10.34 and 32.43 ms with an average of 17.65 ms.

The SES technique is simple and it requires insignificant processing overhead to maintain the state variables required for the estimation process. The technique could be used for reliably estimating CPU requirements for any header or payload processing applications. It can be used for predicting processing requirements of flows having packets with fixed or variable sizes. The SES prediction process can quickly react to a new actual processing time caused by changes in packet size and or processing load/operating system scheduling and can produce a new good estimate for the next packet. It is lot simpler than the linear least square regression technique suggested by Pappu and Wolf [4]. For simplicity and to gain the benefits discussed above, we have implemented the SES technique within our composite scheduler PBFQ for online estimation of execution times of the incoming packets.

4.2. PBFQ algorithm

In a programmable node, packets are first processed in the CPU before they are transmitted through the network link. In this work we have developed a new CPU scheduling algorithm (PBFQ) that can allocate CPU and resources proportionally, adaptively, and fairly among all the competing flows. The algorithm is based on the WFQ principle and has the ability to allocate both CPU resources based on reservations. PBFQ is specially designed for packet scheduling in programmable router environments where traffic patterns can vary significantly and QoS guarantees must be provided to the reserved flows based on their resource reservations.

The node resource manager controls the flow registration and setup (including setting up weights for any reserved flows based on the reserved rates of the bandwidth and processing resources) and admission of each individual packet. The scheduler estimates the processing times of the incoming packets and enqueues them in the corresponding flow queues and dequeues packets using its scheduling algorithm PBFQ, which takes the estimated processing time of packets into account to decide which packet to dequeue. After dequeueing a packet, the scheduler hands the packet to the processor handler object for processing if required. The processor handler object notifies the scheduler after processing each packet so that the scheduler can re-estimate the processing times for the new incoming packets.

PBFQ enforces fairness in resources allocation using the WFQ principle. In traditional WFQ (for bandwidth scheduling) packets from different sessions are stamped with virtual finishing times and selected for scheduling based on increasing order of virtual finishing times. The virtual finish time of a packet indicates a virtual time by which the last bit of the packet must be transmitted through the link. The algorithm uses the packet size and a system virtual time (which is updated each time a new packet arrives) to compute the finishing times of the incoming packets. The virtual time is updated by using a system virtual function and its role is to compute finishing times of packets in newly backlogged sessions in order to equalize the normalized services of these sessions with current backlogged sessions.

However, the WFQ algorithm cannot be directly used for CPU scheduling (as this would require precise knowledge of execution times for packets in advance). Therefore, in PBFQ, we use the prediction techniques to estimate the CPU requirement which is used to calculate the finish time. We define the following equations:

$$\mathrm{sum}_w(t) = \sum_{i \in B(t)} \phi^i_{\mathrm{cpu}} \tag{8}$$

$$V(t+\tau) = V(t) + \frac{\tau}{\mathrm{sum}_w(t)} \tag{9}$$

$$\mathrm{PP}_i(t+\tau) = \alpha P^t_i + (1-\alpha)\mathrm{PP}_i(t) \tag{10}$$

$$S^k_i = \max\{F^{k-1}_i, V(a^k_i)\} \tag{11}$$

$$\mathrm{EP}^k_i = \mathrm{PP}_i(a^k_i) \tag{12}$$

$$F^k_i = S^k_i + \frac{\mathrm{EP}^k_i}{\phi^i_{\mathrm{cpu}}} \tag{13}$$

where

ϕ^i_{cpu}	weights for CPU for flow i
$B(t)$	set of backlogged flows at time t
$\mathrm{sum}_w(t)$	summation of the weights of all the active flows at time t
$V(t)$	system virtual time at time t
τ	time difference between two virtual time updates, i.e. inter-packet arrival time
S^k_i	virtual start time of packet k of flow i
$V(a^k_i)$	system virtual time at the arrival of packet k of flow i
F^k_i, F^{k-1}_i	virtual finish time of packets k and $(k-1)$ of flow i
EP^k_i	estimated processing cost of packet k of flow i in s
P^t_i	actual processing cost of the packet in flow i that is processed at time t
$\mathrm{PP}_i(t)$	predicted processing cost of a packet in flow i at time t
$\mathrm{PP}_i(a^k_i)$	predicted processing cost of a packet in flow i at the arrival of packet k in the flow
α	SES parameter, which was set to 0.5.

The scheduler maintains per-flow variables for storing the values of $PP_i(t)$ and so on. When the scheduler is started, $V(t)$ and $PP_i(t)$ (for all flows) are set to zero. With the arrival of every new packet, the scheduler updates $V(t)$, and calculates the virtual finish time of the packet (F_i^k) using Eqs. (8)–(13). The scheduler then stores the packet in the corresponding flow queue.

While any packet exists within the queues, the scheduler algorithm (i.e. the dequeue process of the scheduler) checks the finish times of the packets at the head of all active flow queues and dequeues the packet (from the head of the queue) having the minimum finish time. It then hands the packet to the processor handler object, which in turn processes the packet and enqueues it in the FIFO transmission queue. After each packet for a flow i is processed, the scheduler updates the predicted processing time for the next packet for the same flow i using Eq. (10).

5. Simulations

5.1. Simulation for SWFQ

5.1.1. Simulation setup

We have implemented SWFQ and SFQ scheme in the Network Simulator Version 2 (NS2) [15]. We simulated three network processing applications that are defined in [4]: IP forwarding with very low processing cost per packet; Cast Encryption with medium processing cost per packet; and forward error coding (FEC) with very high processing cost per packet. The processing requirements of packets for each application have been calculated to approximate real applications on the testbed [4]. The capacity of the server is set to 2000 Mcc/s (Million CPU cycles per second) and processing costs of IP forwarding packet, cast encryption packet, and FEC packet are uniformly distributed at 0.01, 0.116, and 1.90 Mcc, respectively. Simulations are carried out under three schedulers: SFQ, SWFQ, and WFQ. When using WFQ, it is assumed that the processing requirements of packets can be read from the packet headers. In this way, we investigate the trade-off of not knowing execution times when using start time based PFQ algorithms.

5.1.2. Results

- *Fairness and throughput measurement.* We have simulated 10 IP forwarding flows, five cast encryption flows, and two FEC flows with the respective weights for each application flow being 1, 2, and 5. Each flow sends packets at random time interval and the average time interval is set based on the reserved rate. The processing rates are measured by averaging over small time windows which are set about 20 times the average time interval in each application flow. As shown in Fig. 5, the rate plots of the three applications under these schedulers are similar in small time intervals, therefore, it is demonstrated that SFQ and SWFQ have comparable fairness and throughput guarantees to WFQ.

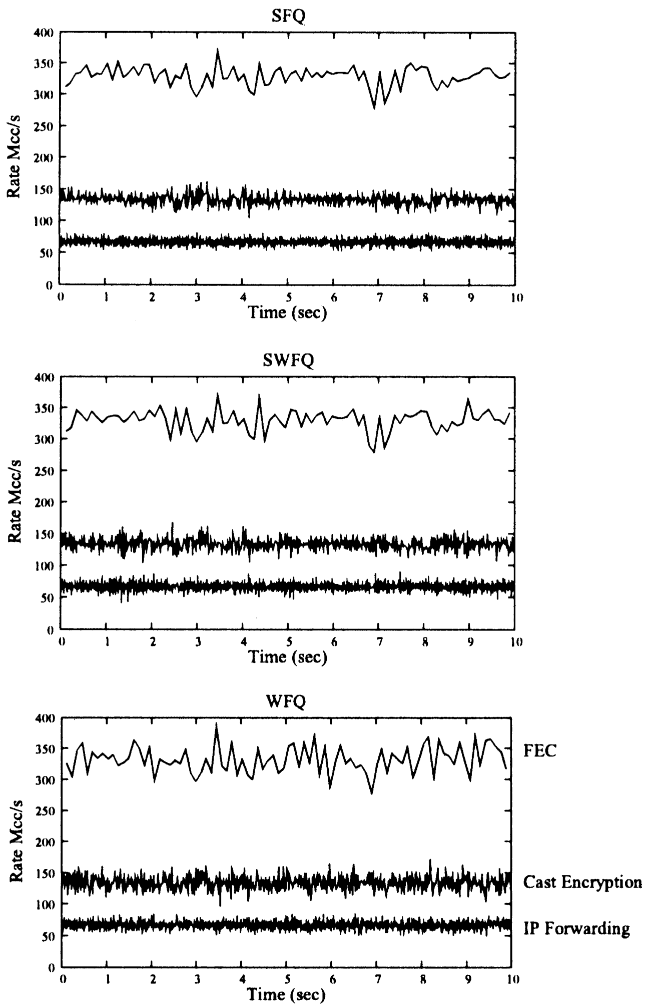


Fig. 5. Processing rates allocated to IP forwading, cast encryption, and FEC.

- *Delay measurement.* We simulated the total of 30 flows, 10 flows for each application. Each flow reserves the same processing rate, and sends packets randomly at specified average time intervals to just saturate its share. The sum of average processing rates of all flows is just under the server capacity. Therefore, the delays measured are mainly due to scheduling not due to queueing backlog. The results in Figs. 6 and 7 show that SWFQ provides lower maximum delays to all application flows, especially, to FEC application flows, when compared with SFQ. As illustrated in Table 1, SWFQ also gives smaller delay standard deviations than SFQ for all applications flows, especially, for IP Forwarding and FEC flows. Reduction in delay standard deviations would reduce the delay jitters. We expect that SWFQ would give better delay behavior due to its more accurate system virtual time, especially, where variations in processing requirements of packets are large (IP forwarding and

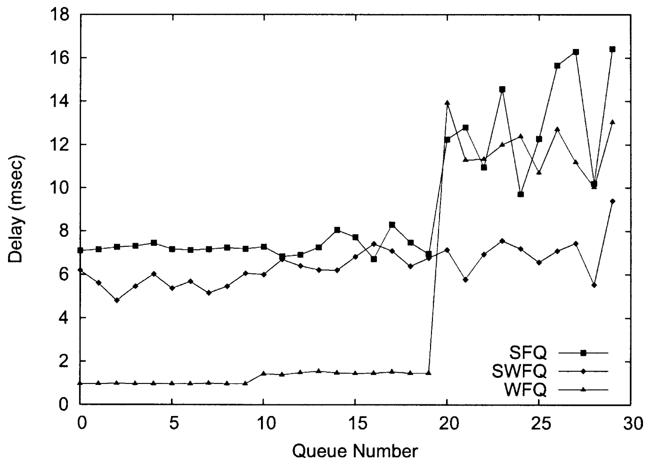


Fig. 6. Maximum delays of packets in IP forwarding (queues: 0–9), cast encryption (queues: 10–19), and FEC (queues: 20–29).

FEC packets). During the processing time of a FEC packet, since the system virtual time in SFQ remains constant, packets arriving at empty session queues during this time can have start times equal to this FEC packet, and thus, are transmitted before packets waiting in backlogged sessions. As a result, packets waiting in backlogged sessions experience larger delays. Fig. 6 shows that WFQ provide smaller maximum delays than SWFQ and SFQ for application flows that have low processing time per packet to reserved rate ratio. However, SWFQ can provide lower maximum delays for FEC packets when compared with WFQ. This behavior is mentioned in the discussion on *delay guarantees* earlier.

5.2. Simulation for PBFQ

In order to demonstrate the characteristics of PBFQ compared to SFQ, we have modified the NS2 network simulator to implement the described programmable node components to simulate a programmable node-based network. The simulation was performed on a PC having a 1.8 GHz Pentium 4 processor with 384 MB memory running under the Linux operating system (RedHat 7.2). The experimental results achieved using PBFQ are compared with the results using SFQ for CPU scheduling. Note that the processing overhead and memory requirement for the proposed system (to classify and manage individual flows) are minimal.

5.2.1. Simulation settings

We used 30 hosts to generate network traffic (UDP packets) for a programmable node. Fig. 8 shows the network topology used for simulation. Hosts 1–10 generated packets containing MPEG2 video data of varying packet sizes and referencing MPEG2 encoder code. Hosts 11–20 generated packets containing encrypted data of varying packet sizes and referencing RC2 decryption algorithm code. The other hosts (21–30) generated packets containing text data of varying packet sizes and referencing RC2 encryption algorithm code. The MPEG2 encoder, RC2 decryption, and RC2 encryption codes were implemented in C++ within the NS2 environment. The simulation settings of the individual flows are given in Table 2.

The output link capacity was set to 10 Mbps. The simulations were run for 300 s and measurements were taken at 3-s intervals. Packet generation rates for all the flows were adjusted such that all the flows required 97% of the CPU resources (i.e. resource utilizations were just below 100%) so that the measured delays are affected by scheduling and not by queueing backlog.

Each simulated NS2 packet header contained some additional fields (or parameters) to specify the referenced code and to facilitate the handling of the packet by the programmable node.

5.2.2. Delay measurements

We measured the delays for all the individual flows for the entire simulation period of 300 s. Figs. 9–11 show the delays measured for an MPEG2 data flow, a decryption data flow, and an encryption data flow using PBFQ and SFQ. The maximum and average delays for these flows are shown in

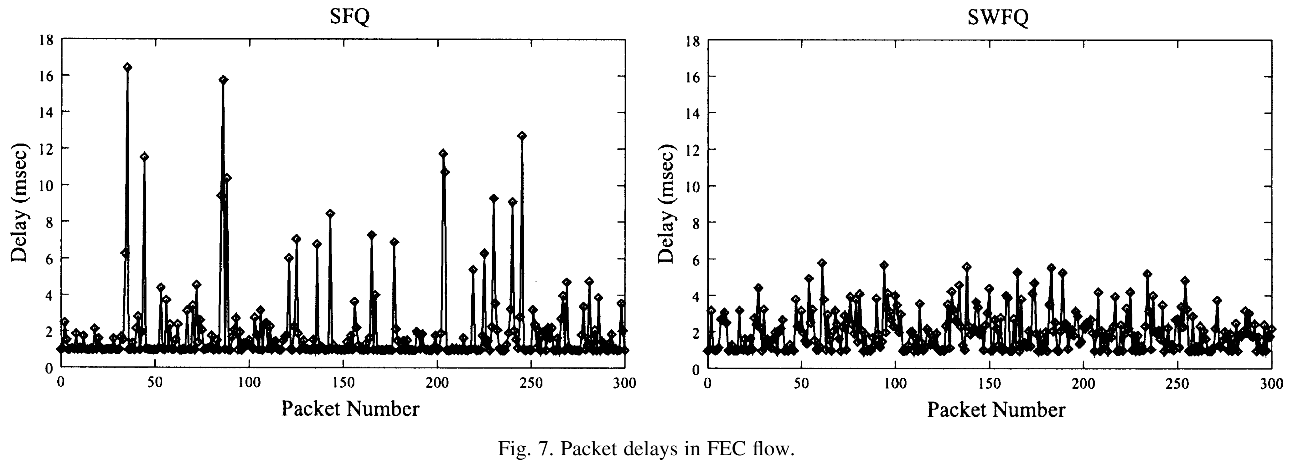


Fig. 7. Packet delays in FEC flow.

Table 1
Delay standard deviations (ms)

Schedulers	IP forward	Cast encryption	FEC
SFQ	1.24	1.27	1.54
SWFQ	0.75	1.08	1.13

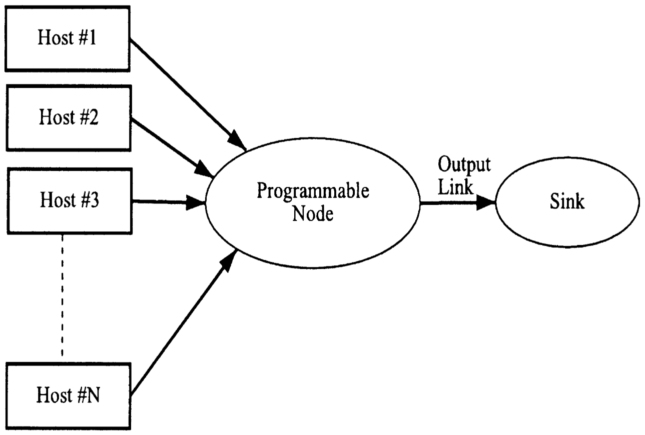


Fig. 8. Network topology used for simulation.

Table 2
Simulation 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	10–40 ms	1–3 ms	1–3 ms
$\phi^i_{\rm cpu}$	3.1	1.0	1.0

Table 3. Delay results show that PBFQ achieved much superior delay guarantees compared to SFQ for all the flows. The results show that using PBFQ, the worst case delay is reduced to 30% for MPEG2 flow, 79% for RC2 decryption data flow and 90% for RC2 encryption data flow compared to the delays achieved using SFQ. Also the average delay was decreased 72% for RC2 decryption data flow and 85% for RC2 encryption data flow compared to the average delays achieved using SFQ. The average delay for MPEG2 flows was better for SFQ (42%). This could be explained as SFQ has tendency to favor flows which have higher average processing time per packet to reserved processing rate ratio [4].

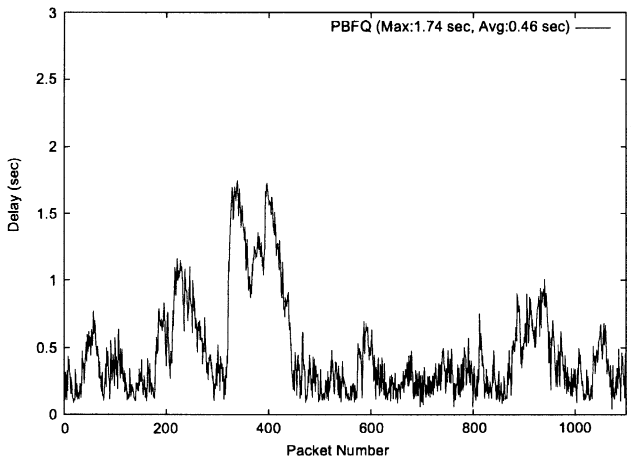
Table 3 also shows the calculated standard deviations of the delays measured. It shows that the PBFQ provides significantly smaller standard deviation of the delay compared to that using SFQ. Using PBFQ the standard deviation reduced to 6% for MPEG2 data flow, 72% for RC2 decryption data flow and 77% for RC2 encryption data flow compared to the standard deviations measured using SFQ. Significantly, smaller standard deviations provided by PBFQ means it provides much more consistent delay guarantees compared to that achieved using SFQ.

5.2.3. Fairness measurements

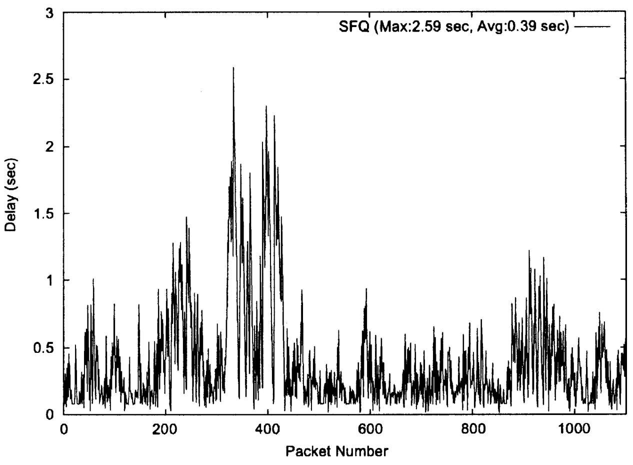
We measured the CPU utilized by all the flows and compared them with the reserved rates (based on the weights used in Table 2) and found that the fairness achieved in utilizing a flow's share of CPU and bandwidth of PBFQ was similar to that achieved by SFQ. Table 4 shows the reserved CPU versus the actual utilized rates for MPEG2, encryption and decryption data flows as measured at the 300th second. Both schedulers allowed a flow to over-utilize its share of CPU when other flows were not sending packets to saturate their shares.

5.2.4. Resource utilization

We measured the CPU utilizations of all the flows every 3 s for the entire simulation period of 300 s and the results are shown in Fig. 12. The results show that both PBFQ and SFQ achieved similar or comparable total CPU utilization.

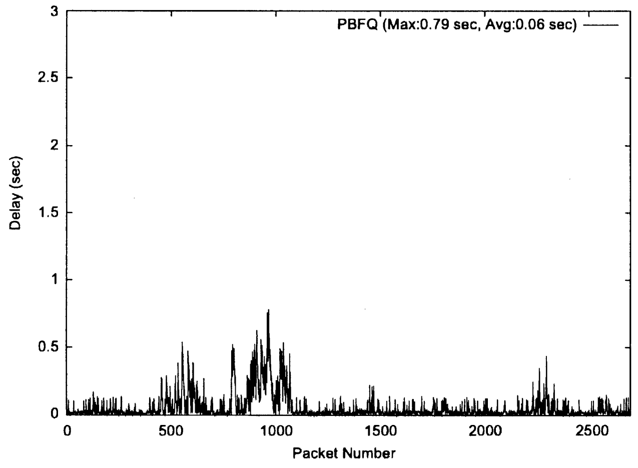


(a) PBFQ

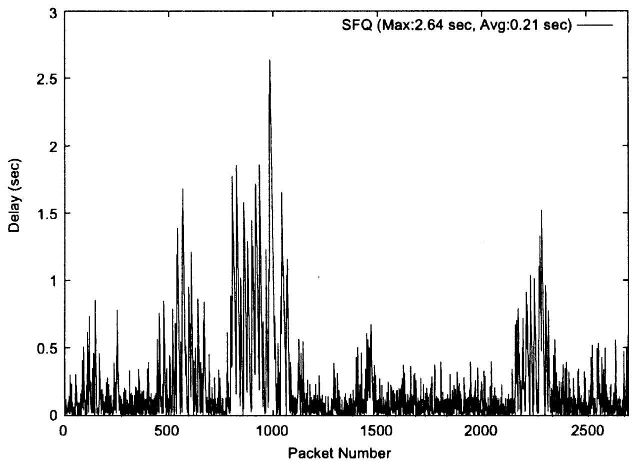


(b) SFQ

Fig. 9. Delay for a MPEG2 data flow.

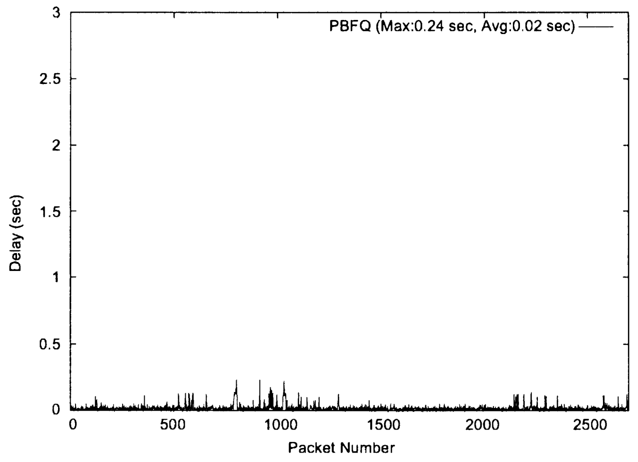


(a) PBFQ

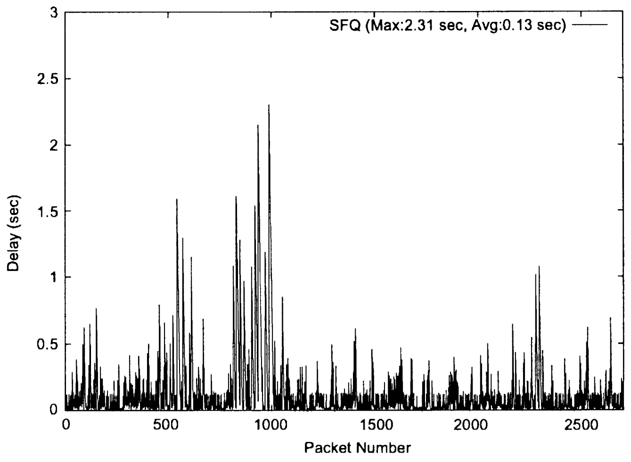


(b) SFQ

Fig. 10. Delay for a RC2 decryption data flow.



(a) PBFQ



(b) SFQ

Fig. 11. Delay for a RC2 encryption data flow.

Table 3
Delay measurements

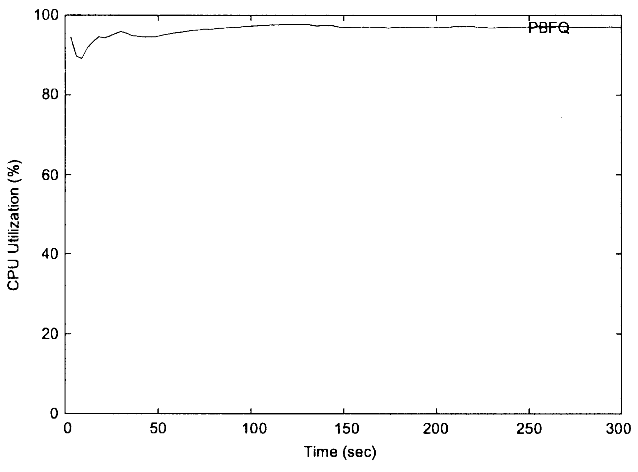
Data flow	Delay using PBFQ (s)			Delay using SFQ (s)		
	Max. delay	Avg. delay	SD	Max. delay	Avg. delay	SD
MPEG2	1.74	0.46	0.36	2.59	0.39	0.38
Decryption	0.79	0.06	0.13	2.64	0.21	0.45
Encryption	0.24	0.02	0.03	2.31	0.13	0.34

6. Conclusion

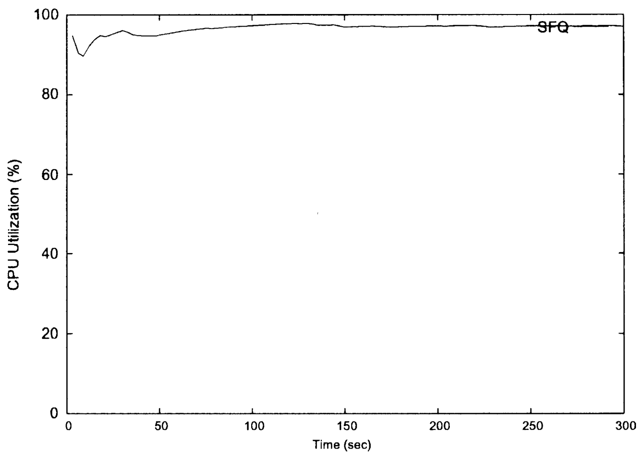
In this paper, we present two novel packet scheduling algorithms called SWFQ and PBFQ. SWFQ does not require each packet processing time in advance for scheduling, where as PBFQ uses the predicted processing requirement of each packet for scheduling. Both of the proposed algorithm could be used for processing resource scheduling in programmable networks to support QoS in two categories: processing resource reservation, and fair

Table 4
Fairness measurements

Data flow	r^i_{cpu}	Utilized CPU rates	
		PBFQ	SFQ
MPEG2	0.06	0.06	0.06
RC2 decryption	0.02	0.019	0.019
RC2 encryption	0.02	0.019	0.019



(a) PBFQ



(b) SFQ

Fig. 12. Total CPU utilization.

processor sharing on a best-effort basis. From analysis and simulation, we show that both scheduling algorithms offer good fairness and delay properties.

Acknowledgements

This research work is supported by Smart Internet Technology Co-operative Research Center (SITCRC), Australia.

Appendix A

Proof of Theorem 1. We use a similar method to the one in [13] to prove the result. Firstly, we prove the following lemma.

Lemma 1. Suppose a packet just finished processing in SWFQ at time t. Then, the minimum start time of packets in current backlogged sessions at time t called S_j is always determined for scheduling and $S_j \leq V(t)$.

This lemma can be proved by contradiction. It is noted that the start time of a session at time t represents the normalize services the session has received up to time t. In the GPS system, all backlogged sessions receive the same normalize service, which is equal to the system virtual time $V(t)$. In SWFQ, by serving packets in increasing order of start times, the system attempts to make the normalize service of each session equal to $V(t)$. If all sessions in SWFQ have start times larger than $V(t)$, it means that all the sessions have been serviced more than it should have in the GPS server. Consequently, the work done in SWFQ will be higher than in GPS, which is impossible since both servers are work-conserving. When Lemma 1 is satisfied, the proof is carried out as follows. Let k be the maximum number of packets in session i, whose service in GPS begins no later than time t. Since packet p_i^{k+1} has not started service in GPS at time t, we have $S_i^{k+1} \geq V(t)$. Let t_1 be the time SWFQ server begins to process packet p_i^{k+1}. From the result of Lemma 1, we have $V(t_1) \geq S_i^{k+1}$. As a result, we have $V(t_1) \geq V(t)$. Since $V(t)$ is an increasing function, it follows that $t_1 \geq t$. Therefore,

$$W_i(0,t) \geq \sum_{j=1}^{k-1} P_i^j \geq \sum_{j=1}^{k} P_i^j - P^{\max}$$
$$= \hat{W}_i(0,t_1) - P^{\max} \geq \hat{W}_i(0,t) - P^{\max} \tag{A1}$$

□

Proof of Theorem 2. We adapt the method of proof in [14] for WFQ to prove the result for SWFQ. As shown in Theorem 1, if each of $(N-1)$ sessions in SWFQ leads GPS by $P^{\max}$, then one session will lag GPS by $(N-1)\,P^{\max}$. This bound is too large for many cases, we can prove a much smaller bound as follows. It is noted that the throughput discrepancy is maximum when a packet starts service in SWFQ. Let t_i^k and $\hat{t}_i^k$ be the time packet p_i^k starts service in GPS and SWFQ, respectively. We only need to consider the case $t_i^k \leq \hat{t}_i^k$; otherwise, $W_i(0,\hat{t}_i^k) \leq \hat{W}_i(0,\hat{t}_i^k)$. Therefore,

$$W_i(0,\hat{t}_i^k) = W_i(0,t_i^k) + W_i(t_i^k,\hat{t}_i^k) \leq \hat{W}_i(0,\hat{t}_i^k)$$
$$+ r_i(V(\hat{t}_i^k) - S_i^k) \tag{A2}$$

At time $\hat{t}_i^k$, since session i receives more service in GPS than in SWFQ, there must be a session j receiving more service in SWFQ than in GPS. Let p_j^m be the packet at the head of session j queue at time $\hat{t}_i^k$. As packet p_j^{m-1} must have finished service before packet k starts service in SWFQ we have $S_j^{m-1} \leq S_i^k$. Since packet p_j^m has not started at time $\hat{t}_i^k$ in SWFQ, and session j is receiving more service in SWFQ than in GPS, we also have $V(\hat{t}_i^k) \leq S_j^m$. Applying these results into (A2), we get,

$$W_i(0,\hat{t}_i^k) - \hat{W}_i(0,\hat{t}_i^k) \le r_i(S_j^m - S_j^{m-1}) \le r_i \frac{P_j^{m-1}}{r_j}$$

$$\le r_i \max_{1\le n\le N}\left(\frac{P_n}{r_n}\right) \tag{A3}$$

Combining the two cases, we have,

$$W_i(0,t) - \hat{W}_i(0,t)$$

$$\le \min\left((N-1)P^{\max}, r_i \max_{1\le n\le N}\left(\frac{P_n}{r_n}\right)\right) \tag{A4}$$

□

Proof of Theorem 3. Let us denote d_i^k and $\hat{d}_i^k$ as the departure times of packet p_i^k in GPS and SWFQ, respectively. We have,

$$\begin{cases} d_i^k = t_i^k + \dfrac{P_i^k}{r_i\rho} \\ \hat{d}_i^k = \hat{t}_i^k + \dfrac{P_i^k}{r} \end{cases} \tag{A5}$$

where $\rho = r/\sum_{i\in B(t)} r_i$, and it represents the ratio of the service rate a session currently receiving to the service rate of that session when all sessions are backlogged in the GPS system.

$$\hat{d}_i^k - d_i^k = (\hat{t}_i^k - t_i^k) + \left(\frac{P_i^k}{r} - \frac{P_i^k}{r_i\rho}\right) \tag{A6}$$

We only need to consider the case $t_i^k \le \hat{t}_i^k$; otherwise, $\hat{d}_i^k \le d_i^k$. Since $W_i(0,t_i^k) = \hat{W}_i(0,\hat{t}_i^k)$, we have,

$$\hat{W}_i(t_i^k,\hat{t}_i^k) = W_i(0,t_i^k) - \hat{W}_i(0,t_i^k) \tag{A7}$$

Combining (A7) with the result of Theorem 2, we have,

$$\hat{W}_i(t_i^k,\hat{t}_i^k) \le \sum_{n\in B(t),n\ne i} (P_n^{\max}) \tag{A8}$$

The worst case delay occurs when the service of session i under SWFQ lag GPS by the maximum amount. As discussed in the proof of Theorem 2, it occurs when all other backlogged sessions in SWFQ lead GPS, only session i in SWFQ lags GPS. Therefore, in interval $(t_i^k,\hat{t}_i^k)$, the normalize service of session i is lower than all other backlogged sessions in SWFQ. Therefore, session i must receive service continuously in this interval. Consequently,

$$\hat{t}_i^k - t_i^k \le \sum_{n\in B(t),n\ne i} \frac{P_n^{\max}}{r} \tag{A9}$$

The delay guarantee for GPS, given in [10], is,

$$d_i^k = L_{\text{GPS}}(p_i^k) \le \text{EAT}(p_i^k) + \frac{P_i^k}{r_i\rho} \tag{A10}$$

Combining Eqs. (A6) (A9) and (A10), we have,

$$L_{\text{SWFQ}}(p_i^k) = \hat{d}_i^k \le \text{EAT}(p_i^k) + \sum_{n\in B(t),n\ne i} \frac{P_n^{\max}}{r} + \frac{P_i^k}{r} \tag{A11}$$

□

References

[1] A. Campbell, H.G. De Meer, M.E. Kounavis, A survey of Programmable Networks. Center for Telecommunication Research, Columbia University, ACM SIGCOMM Computer Communication Review, April 1999.

[2] V. Galtier, K. Mills, Y. Carlinet, Predicting and Controlling Resource Usage in a Heterogeneous Active Network (2001), National Institute of Standards, 2001.

[3] A.K. Parekh, R.G. Gallagher, A generalized processor sharing approach in integrated services networks, In: INFOCOMM'93 1993;.

[4] P. Pappu, T. Wolf, Scheduling Processing Resources in Programmable Routers, Department of Computer Science, Washington University, St Louis, MO, USA, WUCS-01-32, July 25, 2001.

[5] X. Qie, A. Bavier, Larry Peterson, and Scott and Karlin, Scheduling Computations on a Software Based Router. In: Proceedings of the IEEE Joint International Conference on Measurement Modeling of Computer Systems (SIGMETRICS), Cambridge, MA, June 2001.

[6] P. Goyal, H.M. Vin, H. Cheng, Start time fair queuing: a scheduling algorithm for integrated services packet switching networks, IEEE/ACM Transactions on Networking 5 (5) (1997) 690–704.

[7] S.K. Jha, P.A. Wright, M. Fry, Playout Management of Interactive Video—an Adaptive Approach, IWQOS 1997;.

[8] L. Peterson (Ed.), NodeOS Interface Specification, Technical Report, AN Node OS Working group, January 2001.

[9] A. Demers, S. Keshav, S. Shenker, Analysis and simulation of a fair queueing algorithm, Internetworking: Research and Experience 1 (1) (1990) 3–26.

[10] P. Goyal, H.M. Vin, Generalized guaranteed rate scheduling algorithms: a framework, IEEE/ACM Transactions on Networking 5 (4) (1997) 561–571.

[11] G.G. Xie, S.S. Lam, Delay guarantee of virtual clock server, IEEE/ACM Transactions on Networking 3 (6) (1995) 683–689.

[12] J.C.R. Bennett, H. Zhang, Hierarchical packet fair queueing algorithms, IEEE/ACM Transactions on Networking 5 (5) (1997) 675–689.

[13] A.G. Greenberg, N. Madras, How fair is fair queuing?, Journal of the Association for Computing Machinery 39 (3) (1992) 568–598.

[14] J. Rexford, A. Greenberg, F. Bonomi, A fair leaky-bucket shaper for ATM networks, AT&T Report 1995;.

[15] The Network Simulator version 2. Available from www:http://www.isi.edu/nsnam/ns/.