Scheduling Processing Resources in Programmable Routers

Prashanth Pappu and Tilman Wolf
Department of Computer Science
Washington University in St. Louis, MO, USA
{prashant,wolf}@arl.wustl.edu

Abstract—**To provide flexibility in deploying new protocols and services, general-purpose processing engines are being placed in the datapath of routers. Such network processors are typically simple RISC multiprocessors that perform forwarding and custom application processing of packets. The inherent unpredictability of execution time of arbitrary instruction code poses a significant challenge in providing QoS guarantees for data flows that compete for such processing resources in the network. However, we show that network processing workloads are highly regular and predictable. Using estimates of execution times of various applications on packets of given lengths, we provide a method for admission control and QoS scheduling of processing resources. We present a processor scheduling algorithm called Estimation-based Fair Queuing (EFQ) which uses these estimates and provides significantly better delay guarantees than processor scheduling algorithms which do not take packet execution times into consideration.**

I. Introduction

Over the past decade there has been rapid growth in the need for reliable, robust, and high-performance communication networks. This has been driven in large part by the demands of the Internet and general data communications. New protocols, services, standards, and network applications are being developed continuously. However, the ability to deploy these in the current Internet is greatly inhibited by the need for changes in the forwarding loops of routers, which for performance considerations are usually implemented in custom logic. To overcome this obstacle, it has been proposed to place general-purpose processing engines in the data path of routers. Such network processors extend the traditional store-and-forward paradigm to a store-process-and-forward paradigm, which opens vast possibilities for applications like simple quality of service (QoS) forwarding to complex payload transcoding for wireless clients.

In terms of QoS, general purpose processing introduces an additional level of complexity into the system, since not only link bandwidth, but also computational resources have to be shared among packets of competing flows. While a significant amount of work has been done with respect to designing systems, which can provide guaranteed QoS to flows competing for bandwidth, processor sharing poses several new problems in this domain. The problem that we are considering is aimed at routers, where packet processing is performed at the output port. The data path through the output port is shown in Fig. 1. Packets are received from the switch fabric and queued in per-flow queues. Then the processor scheduler assigns packets from the n queues to the m processing engines as they become idle. After processing, packets are again queued in per-flow queues before the link scheduler assigns them to be transmitted on the link. The processor scheduler can view each processing engine as a separate resource to be scheduled if they individually have capacities exceeding the requirements of any single flow. The scheduler can also consider all the processing engines as a single processing resource, which can be scheduled using multi-server variants of single server scheduling algorithms [1]. In either case, the essential problem reduces to designing an efficient scheduling algorithm for sharing a single processing resource.

We provide mechanisms for such a system to give guaranteed bandwidth and computational resources to incoming flows. Guarantees in these two *dimensions* mean that a flow always gets its reserved shares except when:
1. A flow requires computational resources in excess of its reserved capacity and hence only a fraction of the incoming traffic is processed and forwarded to the link scheduler, possibly giving the flow a lesser share of its reserved bandwidth.
2. Or equivalently, a flow exceeds its link share resulting in too many packets being queued up at the link scheduler, which forces the processor scheduler not to give the flow its processing share.

Realizing such a system is fundamentally complicated by the fact that the execution times of various applications on packets are not known in advance, which limits applicability of well known bandwidth scheduling algorithms. Also, at a flow level, it is not clear as to how explicit or implicit admission control can be done as the processing requirements of a single flow are not known.

In this paper, we first present actual execution times of various applications on packets of varying lengths, measured on a programmable router. We show that for the restricted class of network applications, the processing times are strongly correlated to the size of the data being processed (i.e., packet length). We then use this correlation to predict packet execution times to perform admission control and to schedule packets for processing. We present a scheduling algorithm called Estimation-based Fair Queuing (EFQ), which unlike bandwidth schedulers uses the estimates of packet execution times and provides better delay bounds than processor scheduling algorithms which do not use packet execution times at all.

The paper is organized as follows: Section II discusses related work. Section III demonstrates the predictability of packet processing times and shows how admission control on processing resources can be done. Section IV describes the scheduling algorithm EFQ in detail and Section V presents the simulation results. Conclusions are drawn in Section VI.

II. Related Work

A significant amount of work has been done in defining architectures for software based programmable routers [2] [3] [4]. In particular, we have extensively used the router plugins architecture in our work [5] [4]. Most systems enforce isolation of packet processing between flows (e.g., malicious packets cannot effect the proper processing of other packets). However, QoS issues at the level of processing are addressed only in a few cases. The commonly used NodeOS specification [6] asks for packets

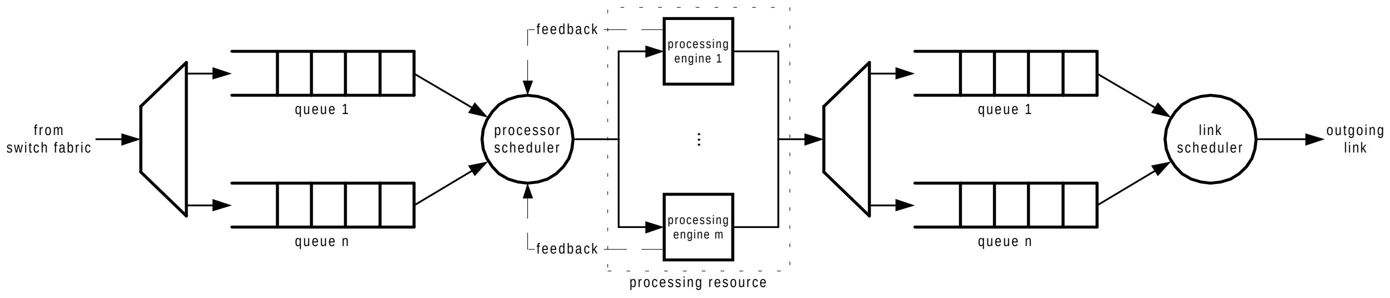


Fig. 1. System Outline.

to be processed by individual threads to allow for an accounting mechanism. However, methods for admission control and QoS scheduling are not described. Reference [7] describes the problem of scheduling computational resources among competing flows, but relies on being able to pre-determine the processing time of packets. Also the more important issue of correlating the cycle rate of a flow to the bit rate is not addressed. There are also approaches where the expressiveness of the processing environment is restricted (e.g, no loops) to give execution time guarantees [8], which limits the usefulness to simple header processing applications.

Packet service disciplines and their associated performance issues have been widely studied in the context of bandwidth scheduling in packet-switched networks [9]. The performance of these disciplines has been compared to Generalized Processor Sharing (GPS) [10], which has been considered an ideal scheduling discipline based on its end-to-end delay bounds and fairness properties. Packet Fair Queuing (PFQ) disciplines, however, cannot be used for processor scheduling. PFQ disciplines like WFQ, WF^2Q [11] use a notion of virtual time, whose correct update in a processor scheduler, requires precise knowledge of execution times of various packets *in advance*. Efforts have been made to design service disciplines which isolate the scheduler properties that give rise to ideal fairness and delay behavior, without emulating GPS [12]. Notable among these are a class of schedulers called Rate Proportional Servers [13], which decouple the update of system virtual time from the finish times of packets in queues. But even these service disciplines, while avoiding the complexity of GPS emulation, schedule packets in order of pre-determined finish times, which in turn requires the knowledge of execution times of various packets in advance.

An exception to these disciplines is Start-Time Fair Queuing (SFQ) [14], which has been deemed suitable for CPU scheduling [15]. Since SFQ does not need prior knowledge of the execution times of packets (packet lengths in a bandwidth scheduler), it is also applicable to scheduling computational resources. However, the worst case delay under SFQ increases with the number of flows and can in fact worsen in the presence of correlated cross-traffic as shown in [16]. As we will show in later sections, SFQ tends to favor (provide lesser queuing delays) to flows which have a higher average processing time per packet to reserved processing rate ratio.

Our work is aimed at providing a way of estimating execution times of packets, which is used on a flow level for admission control and for QoS scheduling at a packet level.

III. Reservation of Processing Resources

A key component of quality of service is the definition of the service that is requested by a flow. While this is straightforward and well understood for link resources, reservations for computational resources are not as clearly defined. This comes from the unpredictability of general purpose processing. In principle, the halting problem states that it cannot be determined if an arbitrary program ever terminates. Thus, the execution time of an arbitrary piece of instruction code cannot be determined, in particular, when the execution time depends on data fields in the packet.

However, networking applications often require very regular, predictable processing. Our measurements, which are discussed below, indicate that for certain application classes, the processing times are very tightly correlated to the packet size. This holds true on a per-flow granularity, where processing requirements are dependent on the flow bandwidth, as well as on a per-packet granularity, where the processing time is dependent on the packet size. This correlation can be exploited to *predict* processing requirements of packets and flows and use the prediction for admission control and scheduling.

A. Predictability of Processing Requirements

A.1 Application Types

Applications that process packets on routers can be divided into two categories: header-processing applications and payload-processing applications [17]. Header-processing applications are characterized by the fact that the processing of the packet is restricted to read and write operations in the header of the packet. This means that the processing complexity is in general independent of the size of the packet. Examples of header-processing applications are IP forwarding, transport layer classification, and QoS routing. Payload-processing applications, in contrast, are characterized by read and write operations to all the data in the packet, in particular, the payload of the packet. It is here that the processing complexity strongly correlates to the packet size. Typically, payload processing applications also show a header-processing overhead in addition to the payload processing. Examples of payload-processing applications are IPSec encryption, packet compression, and packet content transcoding (e.g., image format transcoding).

A.2 Measurements

We have measured the processing times for four applications: IP forwarding, which is a header-processing application, and encryption (CAST), compression (Adaptive Huffman Coding), and forward error correction (Reed-Solomon), which are payload-processing applications. The packet processing times were acquired using a programmable line card [18] on the Washington University Gigabit Router [19]. Processing was performed in the Crossbow [5]/ANN [4] operating system.

The measured results over a range of packet sizes are shown in Fig. 2. The average processing times are shown as lines and the error bars indicate the range of the 95% percentile of processing times. Note that we use time as the unit for processing cost. This is done to simplify the description of the scheduling algorithm and its analysis. In a realistic network, processing cost should be translated to processor cycles per second and then adapted to the particular router system, where the packets get processed, as described in [20].

For IP forwarding, the processing time is practically constant for all packet sizes, which shows the per-packet processing cost of header processing. However, the processing times of the three payload processing applications are clearly dependent on the packet size. The per packet processing time for these applications can be extrapolated for packets of size 0. With these observations, we can approximate the processing cost c of a packet of length l when processed by application a as

$$c = \alpha_a + \beta_a \cdot l, \tag{1}$$

where α_a is the per packet processing cost and β_a is the per byte processing cost of application a. Thus, the processing requirements of these applications can then be described by two parameters: α_a and β_a. These parameters for the three applications are shown in Table I.

A.3 Online Estimation

Though the parameters α_a and β_a have been determined from traces, given this strong correlation between packet sizes and execution times, it is possible to determine these parameters online and in fact improve them, using simple linear least squares regression techniques. As packets are processed the router maintains variables denoting the sums, $\sum c_i, \sum l_i, \sum c_i^2, \sum l_i^2, \sum (c_i \cdot l_i)$ for each application a. These variables are updated on the arrival of a new (c_{n+1}, l_{n+1}) pair. The parameters to be used in the estimation can then be computed as

$$\beta_a = \frac{\sum_n c_i \cdot l_i - \sum_n c_i \cdot \sum_n l_i / n}{\sum_n l_i^2 - \sum_n l_i \cdot \sum_n l_i / n}, \tag{2}$$

$$\alpha_a = \sum_n c_i - \beta_a \cdot \sum_n l_i. \tag{3}$$

It should be noted that there are also applications, where the processing time cannot be as nicely correlated to packet size as shown above. An example for such an application is MPEG encoding. For MPEG encoding a whole video frame is required to perform effective compression. With unencoded video frames typically exceeding a packet size, processing can only be performed once several packets of a flow are buffered. In this case the processing time varies significantly between packets, but it can be expected to be more evenly distributed over frames (i.e., I-frame to I-frame). In such a case the parameters should be maintained for the group of packets constituting a single frame, which are always processed together.

B. Bandwidth Expansion

Processing of packets on routers can affect the size of the packets after the processing is completed. For many types of applications (e.g., encryption, routing lookup) the packet size is not changed, but a few applications can significantly change the bandwidth of a flow (e.g., compression, FEC). To take these changes into account, we define an expansion factor, γ_a, that is the average output bandwidth divided by the input bandwidth. This factor is also shown in Table I. Note that the expansion factor can be dependent on packet size and data as for the compression application.

C. Admission Control

In an environment, where we want to be able to give service guarantees to data flows, it is typically necessary to explicitly reserve resources for that flow. This happens during the flow setup and allows the network to route a new flow in such a fashion that enough bandwidth is available on the chosen path. Now that we have shown that the processing requirements for a stream of data can be described in a simple manner, we can integrate this information into the flow setup process.

A reservation for a flow j with incoming bandwidth B_j that is processed by application a needs to reserve $\gamma_a \cdot B_j$ bandwidth on the outgoing link. The amount of processing P_j that is required (as fraction of one processor) depends on the bandwidth of the flow, the average size of packets l_j, and the application parameters:

$$P_j = \frac{B_j \cdot c}{l} = \frac{B_j}{l} \cdot (\alpha_a + \beta_a \cdot l). \tag{4}$$

Thus, flow j can be admitted to any router that has P_j processing power and $\gamma_a \cdot B_j$ outgoing bandwidth available. How an efficient or optimal route can be found with these parameters is outside the scope of this paper. In principle, an optimal route can be found by combining processing and transmission costs into one metric [21]. Another approach is to aggregate processing availability information together with bandwidth and topology data similar to PNNI [22].

IV. Processor Scheduling

The choice of the packet service discipline for scheduling the processing resources is an important issue in guaranteeing end-to-end delay bounds and ensuring fair sharing of processors among competing flows.

We note that the exact processing time of an application on a packet of a given size cannot be pre-determined and hence precludes the use of many well known packet scheduling algorithms. However, we can use a *good* estimate of the execution time using parameters obtained in Section III for designing a scheduling algorithm that has good delay and fairness properties. In this section we describe how we build upon the class of rate-proportional servers, which have desirable properties that allow the use of these estimates to design a processor scheduling algorithm called Estimation-based Fair Queuing (EFQ).

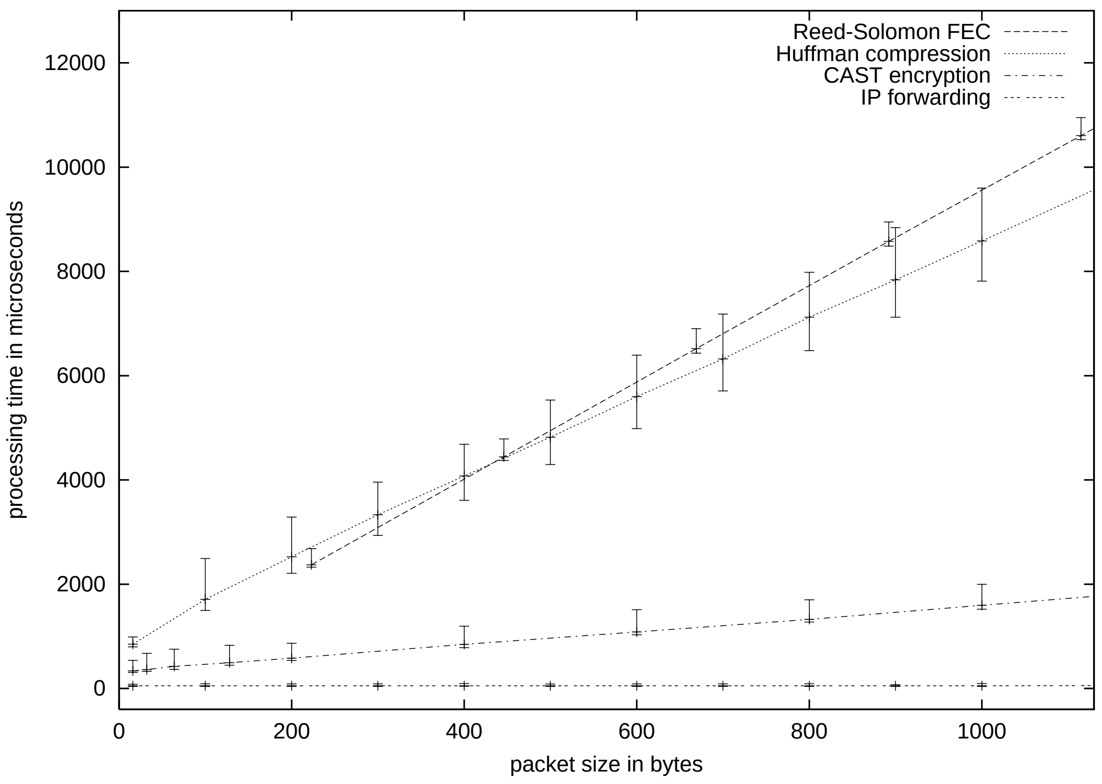


Fig. 2. Processing Times of Programmable Router Applications. The error bars indicate the 95%-percentile of processing times.

TABLE I
PACKET PROCESSING PARAMETERS.

Application a	per-packet cost α_a (μs per packet)	per-byte cost β_a (μs per byte)	expansion factor γ_a
IP forwarding	51	0	1
Encryption	320	1.3	1
Compression	970	7.6	0.13 - 0.34
FEC coding	320	9.2	1.14

A. Rate Proportional Servers

A.1 Definition

Rate Proportional Servers (RPS) are a class of scheduling algorithms designed according to the methodology presented in [13], which allows the designer to trade fairness of the algorithm with implementation complexity. Generally speaking, a rate-proportional server is a work-conserving server with the following properties:

1. The server has an associated system potential, which is updated to reflect the total work done by the server.
2. Each flow in the system has an associated potential. When a flow becomes backlogged, its potential is set equal to the system potential. When a flow is already backlogged, its potential is updated to reflect the normalized service received from the server.

By imposing conditions for the potential functions as given in [13] and by serving packets from flows such that at any instant the individual potentials of all backlogged flows are equal, it can be shown that rate proportional servers have delay and fairness properties comparable to GPS. WF^{2}Q+ [16] is an important example of a scheduler belonging to the RPS class.

We build on this methodology in designing the EFQ processor scheduling algorithm for two important reasons. First, the methodology helps in designing algorithms with delay bounds and fairness comparable to GPS without the complexity of GPS emulation. More importantly, the methodology provides us with enough flexibility to decouple the update of system potential from the exact finish times of the packets in the queues, which addresses the problem of not knowing the exact processing times in advance.

A.2 Packet Selection Policy

A scheduling algorithm with optimal fairness would have to schedule single processing cycles according to the fluid Rate Proportional Server. However, in network processors, the smallest unit of processing is a complete packet. Context switching between packets is not considered here, because saving and recovering processing state is a relatively expensive operation compared to the short overall processing time for a packet.

Thus, to approximate a fluid RPS, packets should be scheduled in order of their finish time with the earliest finish time first. While this works perfectly fine for bandwidth schedulers, the lack of the knowledge of the actual execution times of the packets, makes an exact implementation infeasible for processor schedulers.

However, to derive an approximate scheduler of this class, we can generalize the definition of a packet-by-packet RPS. Such a scheduler schedules two packets, j and k, of flows A and B, in the order in which they are *more likely* to finish processing, i.e., if F_a^j and F_b^k are random variables representing the finish times of these packets in the fluid RPS, then packet j is scheduled for service before k, if

$$P(F_a^j \geq F_b^k) \geq 0.5. \tag{5}$$

Hence, it is the knowledge of the distributions of F_a^j and F_b^k which determines the accuracy with which schedulers can approximate GPS even if they use the same potential (or virtual time) functions. Also, since the potentials of individual flows are updated according to the normalized service received by the flows from the system, the finish time F_a^j is

$$F_a^j = P_a + \frac{W_a^j}{R_a}, \tag{6}$$

where P_a is the potential and R_a is the rate of service reserved by flow A. While these are known in advance when determining F_a^j, W_a^j, which represents the service time required by packet j, is not. Thus, the random variable F_a^j is directly determined by W_a^j.

Start-time Fair Queuing (SFQ) [14] (with a modified system virtual time) and WF^{2}Q+ [16] are scheduling algorithms belonging to this class that represent the extremes with respect to the amount of knowledge of F_a^j. SFQ does not use any information about the service time of a packet and hence, according to the above policy, SFQ schedules packets in increasing order of P_a, which makes it suitable for processor scheduling. WF^{2}Q+, on the other hand, assumes that the exact service times of all packets are known in advance and thus determines the right order of servicing packets with probability 1.

A.3 Misordering Delay

Different schedulers using the same potential functions and ordering packets for execution according to the above defined policy can give varying delays to flows based on their knowledge of the random variables W_a^j. To quantify these delays, assume that a scheduler of this class can be characterized by random variables χ_{a^j,b^k}, which denote the event that the scheduler (with its knowledge of W_a^j and W_b^k) makes a mistake in ordering packets j and k. I.e., $P[\chi_{a^j,b^k} = 0]$ is the probability that the scheduler orders the packets of these two flows *correctly*, while $P[\chi_{a^j,b^k} = 1]$ is the probability that the scheduler makes a *mistake* in the ordering. Then, the average misordering delay, δ_a, as seen by a packet of flow A is the additional delay caused by the scheduler misordering packets of flow A and flow B, which is

$$\delta_a = P[\chi_{a^j,b^k} = 1] \cdot \frac{R_b}{R} \cdot (P_b + \frac{W_b^j}{R_b} - P_a - \frac{W_a^i}{R_a}). \tag{7}$$

This accounts for the time spent by the server in servicing *additional* traffic from flow B before processing packet from flow A. It is these additional delays caused by misordering of packets that we intend to reduce using the estimates of the packet execution times we derived in Section III which improves the schedulers knowledge of W_a^j.

B. Estimation-Based Fair Queuing

Estimation based Fair Queuing (EFQ) is a scheduling discipline designed for processor schedulers that uses the estimates of the packet execution times in ordering packets of various flows for processing. While the packet selection policy of any Rate Proportional Server can be changed to use these estimates, EFQ is derived by modifying WF^{2}Q+ which is known to have the tightest delay bounds and low time-complexity among bandwidth schedulers.

EFQ, like WF^{2}Q+, uses a notion of system virtual time (system potential), defined by

$$V(t+\tau) = max(V(t)+\tau, min_{i\epsilon B(t+\tau)} S_i), \tag{8}$$

where B(t) represents the set of backlogged flows at time t and S_i the start-tag associated with flow i as defined below. The above definition of $V(t)$ makes WF^{2}Q+ a Rate-Proportional Server. It differs from SFQ, in that it has a linear component, which ensures that the delay bounds provided are within one packet servicing time of a corresponding GPS server [16].

For each flow i in the system, EFQ maintains a start tag, S_i (potential of flow i), a finish tag, F_i, and an estimated finish time tag, EF_i. Consider a packet k of flow i, with a reserved rate r_i, that arrives at time a_i^k. When this packet reaches the head of the queue, S_i is updated using

$$S_i = \max(F_i, V(a_i^k)), \tag{9}$$

if queue i is empty, else

$$S_i = F_i. \tag{10}$$

EF_i is updated using

$$EF_i = S_i + \frac{E_i^k}{r_i}, \tag{11}$$

where E_i^k is the estimated number of instructions required to process packet k. This estimate is derived from the length of the packet L_i^k and the parameters α_a and β_a of the application processing the flow using Equation 1:

$$E_i^k = \alpha_a + \beta_a \cdot L_i^k. \tag{12}$$

When the processor finishes processing this packet, the actual finish tag F_i is updated using feedback from the processor:

$$F_i = S_i + \frac{A_i^k}{r_i}, \tag{13}$$

where A_i^k is the actual number of instructions required to process packet k. This ensures that each flow is correctly *charged* for processing time, even if the initial estimate was incorrect.

Given these tags, the EFQ scheduler, schedules packets in increasing order of their estimated finish time tags EF_i.

C. *Example*

The following illustrates the behavior of EFQ and compares it to that of SFQ and WF^{2}Q+. Consider a set of flows, all of which send packets of the same length but at different rates and are processed by the same application. Fig. 3 shows six such flows, with flow 1 reserving 50% of the processing resource and the rest of the flows reserving 10% each. The size of a packet in Fig. 3 represents the *actual* processing time of that packet. Note, however, that the *estimates* for all packets are the equal, since they all have the same length and are processed by the same application.

WF^{2}Q+ achieves an optimally fair schedule, because it is assumed the scheduler knows the actual processing times. Thus, the packets of flow 1 and the other flows alternate (due to the rate reservations). Out of flows 2-6, the packet of flow 2 is processed first, because it has the lowest actual execution time and therefore the lowest finish time.

EFQ *expects* all packets to have the same execution times. Thus, EFQ could pick any order of packets 2-6 to alternate with packets from flow 1. The worst case, which introduces most misordering delay, is shown in Fig. 3. Here, the packet of flow 2 is processed after packets of flows 6,5,4 and 3 are processed, which all use more processing time than expected by scheduler. As a result, the packet from flow 2 experiences an additional delay due to the variation in actual processing times of these packets. However, these variations are much smaller (and bounded, for the applications in consideration) than the total processing times of the packets themselves. In particular, these delays are much smaller than those introduced by SFQ.

As shown in the example, in the worst case SFQ could delay the processing of the first packet of flow 1 until packets from all other flows are processed. This is due to all initial packets having the same start time.

In summary, EFQ processes most packets in the same order as WF^{2}Q+. When either a flow reserves a much higher rate than others or has greatly differing processing requirements (due to differing packet sizes or applications), the variations in the actual executions times compared to estimated execution times do not change the scheduling order. Even in the case when the scheduling order of packets in EFQ varies from that of WF^{2}Q+, the additional delay that is experienced by a packet is bounded by the *variation* in execution times as opposed to the *total* execution times of packets as in SFQ.

D. *Analysis*

From the example given above, it can be seen that for N flows, in the worst case, SFQ introduces a misordering delay of

$$\delta_{SFQ} = \sum_{i=1}^{N} \frac{A_i^{max}}{R} - \frac{A_a^{max}}{R_a}. \tag{14}$$

This is obtained by using $\forall k : \chi_{a^j,b^k} = 1$ with the misordered packets being of maximum size and using $\forall b : P_b = P_a$ in Equation 7, since the scheduler can make a mistake only when $P_b \leq P_a$. Results in Section V also show that SFQ actually favors (i.e., gives lesser delays to) flows with packets which require greater average normalized service (i.e., higher $\frac{E_a^{avg}}{R_a}$).

To analyze EFQ, assume that for a given packet length, the packet execution time estimates obtained in section III can be represented by uniform random variables W_a^j lying in the range $[E_a^j - V_a^j, E_a^j + V_a^j]$. The EFQ scheduler misorders packet j and k when it determines that

$$P_a + \frac{E_a^j}{R_a} \geq P_b + \frac{E_b^k}{R_b}, \tag{15}$$

but the actual processing times are such that

$$P_a + \frac{A_a^j}{R_a} \leq P_b + \frac{A_b^k}{R_b}. \tag{16}$$

In the worst case, we get

$$\frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} \leq P_b - P_a \leq \frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} + \frac{V_a^{max}}{R_a} + \frac{V_b^{max}}{R_b}. \tag{17}$$

Hence from Equation 7, the misordering delay for packet j due to packet k is limited to

$$\delta_a = P[\chi_{a^j,b^k} = 1] \cdot \frac{R_b}{R} \cdot (\frac{V_b^{max}}{R_b} + \frac{V_a^{max}}{R_a}) \tag{18}$$

and the worst case misordering delay is bounded by

$$\delta_{EFQ} = \sum_{i=1}^{N-1} \frac{V_i^{max}}{R} - \frac{V_a^{max}}{R} + \frac{V_a^{max}}{R_a}. \tag{19}$$

From the above equation we can see that as the number of flows increases, δ_{EFQ} only increases with the variations in execution times as opposed to δ_{SFQ} which increases with total processing times. Also note that, with a better estimation, e.g., by including higher order moments in characterizing W_a^j, EFQ can more accurately determine the right scheduling order, resulting in a smaller δ_{EFQ} and thus approximating WF^{2}Q+.

V. Simulation Experiments

In this section, we present simulation experiments to demonstrate the improved performance of EFQ as compared to SFQ.

A. *Simulation Setup*

To compare the delay characteristics of the two schedulers, we use the following simulation setup. First, we obtain traces of the actual execution times of packets from different flows that are processed by different applications on the programmable router. These traces are then used by a packet generator to feed the two simulated schedulers: SFQ and EFQ. The speed of the processor in the simulator is 2GHz (about 10 times the speed of the processor on the Smart Port Card (SPC) [18] on which the actual measurements were made). The system has 32 flows with different packet sizes, which are processed by the four different applications. All the flows reserve the same procesing rate and adjust their sending rates to just saturate their share of the processing resource. These flows together require just below 100% of the system's processing resources. Thus, they can all be admitted and the measured delays are only due to scheduling and not due to queuing backlog.

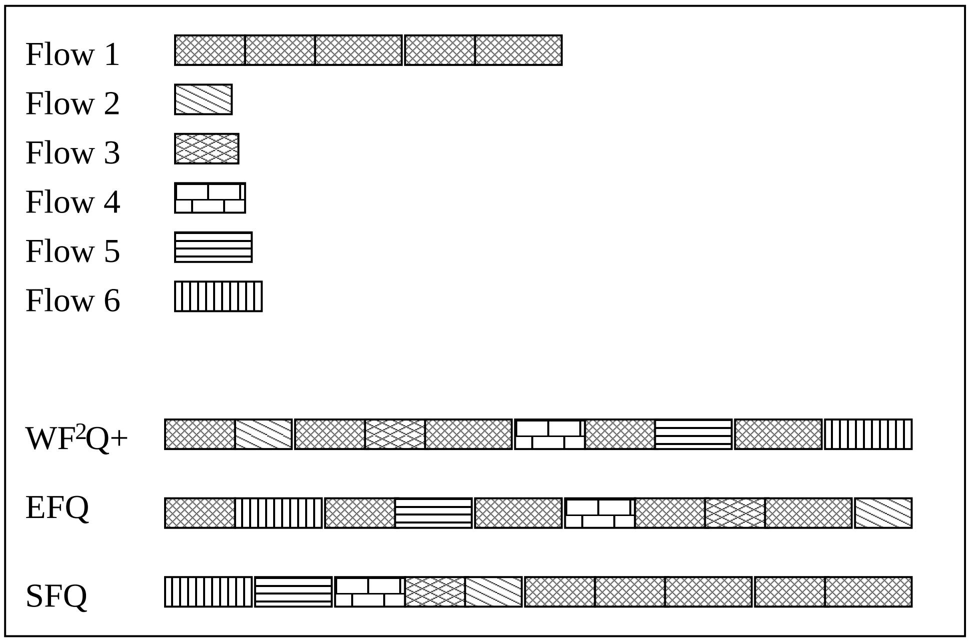


Fig. 3. Scheduling Example. All flows have backlogged packets of the same length and are processed by the same application. The figure shows the actual execution times of packets as their size and the processing order derived by different scheduling disciplines.

B. *Delay Plots*

Fig. 4 shows the delays of various packets of a flow, which is processed by the forwarding application. The interarrival time of the packets of the flow is approximately 163 microseconds, which is just enough to saturate the flow's share of processing resources. Note the high and bursty delays experienced by the packets of the flow when scheduled by SFQ as shown in Fig. 4(a). Since SFQ, always schedules packets with the minimum virtual time, a single packet of a flow can be delayed in the worst case by the equivalent of the sum of one packet processing time of *all* other flows. In the simulation this translates to a worst case misordering delay of 8218 microseconds. The maximum delay actually observed in Fig. 4(a) is about 6100 microseconds, implying an observed *maximum* misordering delay of $6100 - 163 = 5937$ microseconds.

For EFQ, much lower delays can be seen in Fig. 4(b). This illustrates two things. Firstly, given the small execution time of forwarding as compared to other applications, the finish times of the packets of this flow where so different compared to the finish times of the packets of other flows that the errors in estimates did not change the scheduling order (i.e., Equation 17 was not satisfied for most comparisons of finish times). Secondly, the worst case delay that could be experienced by these packets is only 1312 microseconds which would occur if there were maximum variations in the estimated execution times for packets from all other flows at the same time. In the simulation, the *maximum* misordering delay observed is about $900 - 163 = 737$ microseconds.

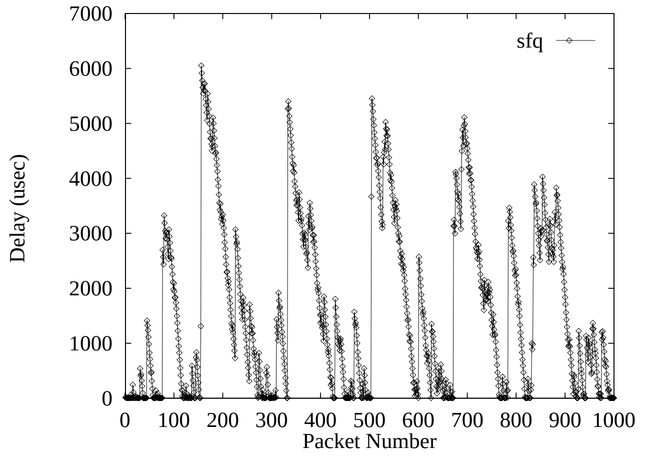
Fig. 5 shows the delays experienced by a flow being processed by the CAST encryption application, with the average packet size of the flow being 200 bytes and has a higher average processing time per packet compared to the forwarded flow. While the *average* delays experienced by the packets when scheduled using EFQ is close to the interarrival time of the packets indicating a very low misordering delay, the *average* delays seen in Fig. 5(a) are about *thrice* the interarrival time of the packets. Fig. 6 shows the delays experienced by a flow being processed by the FEC application which requires much greater processing time per packet compared to the above flows. Here, the *average* delays seen by the packets when scheduled by SFQ are actually *less* than the interarrival time of the packets! indicating an average negative misordering delay, while those due to EFQ are are just about the interarrival time of the packets.

Two important conclusions can be drawn from these plots:
1. SFQ gives much higher misordering delay bounds than EFQ.
2. Across flows, while the misordering delays due to EFQ are on an average close to zero, they vary from high positve misordering delays (e.g., the delay of about 35 times the interarrival rate seen by the forwarding flow) to low negative misordering delays when scheduled using SFQ.

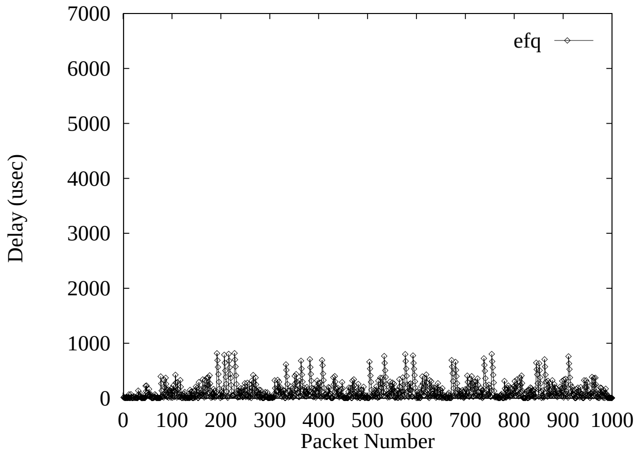
C. *Biased Delay Bounds Due To SFQ*

The second conclusion can be explained by the work conserving nature of the two schedulers. If SFQ gives high positive misordering delays to some flows, there should be flows in the system which get low and in fact negative misordering delays, while EFQ gives low (close to zero) average misordering delays for all flows. We actually show a correlation between the misordering delay experienced by the packets of a flow and the average processing time per packet to reserved processing rate ratio (i.e., $\frac{E_a^{avg}}{R_a}$).

SFQ favors and gives less misordering delays to flows with higher average processing time to reserved rate ratio over flows with a lower ratio. Given a set of flows with the same potential, since SFQ can schedule them in any random order, it is very likely that a packet of a flow with higher average processing time to reserved rate ratio is scheduled before at least a *few* flows

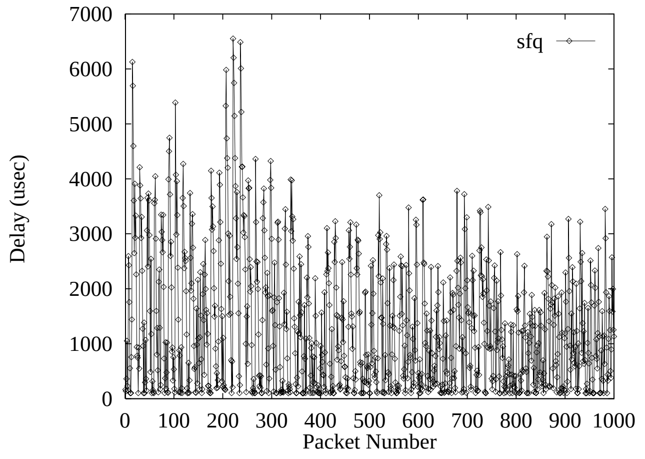


(a) SFQ

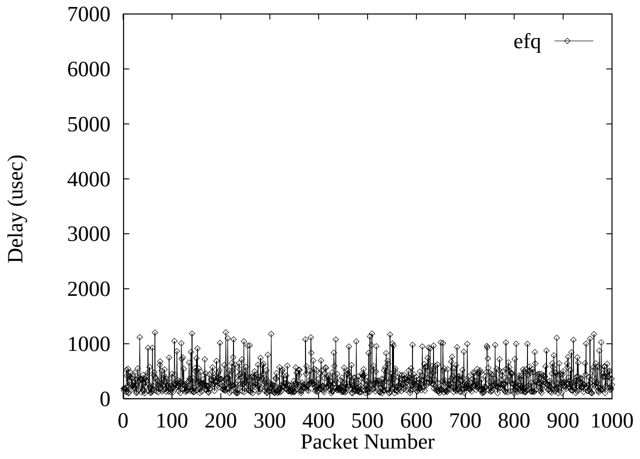


(b) EFQ

Fig. 4. Packet Delays for a Flow Processed by IP forwarding.

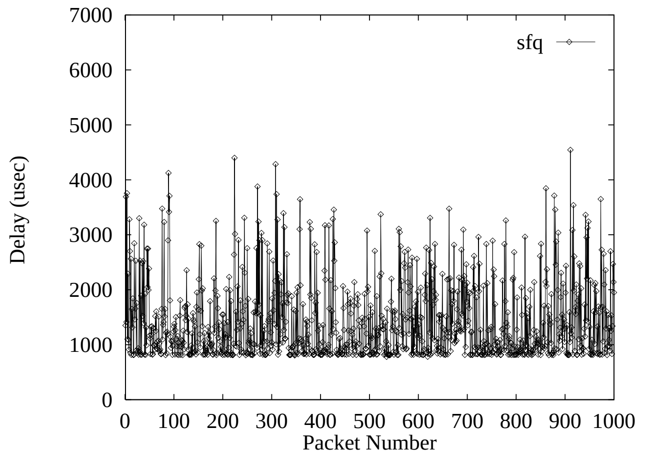


(a) SFQ

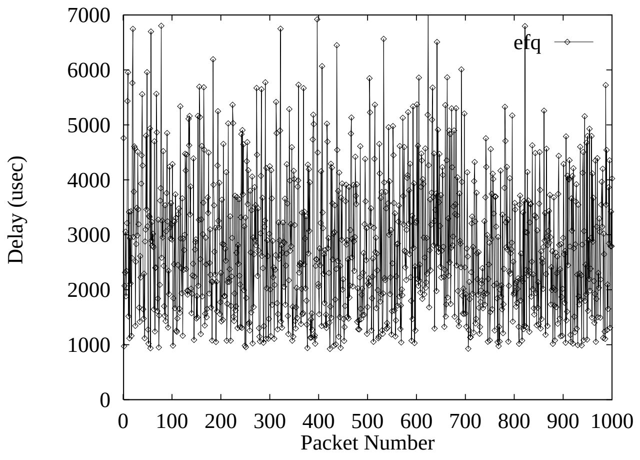


(b) EFQ

Fig. 5. Packet Delays for a Flow Processed by CAST Encryption.



(a) SFQ



(b) EFQ

Fig. 6. Packet Delays for a Flow Processed by Reed-Solomon FEC.

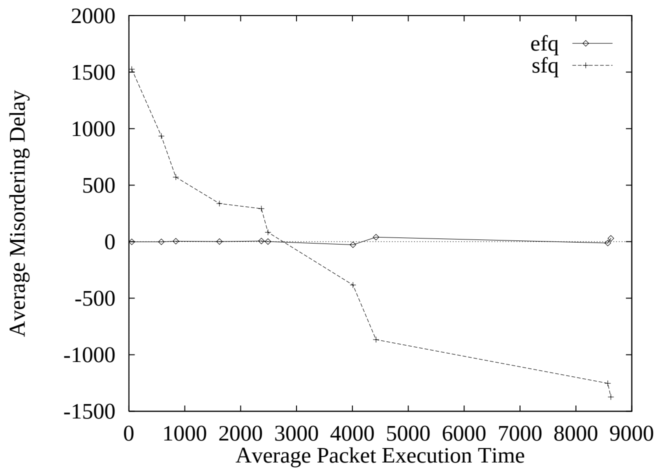


Fig. 7. Variation in Minimum Packet Delay for Different Flows Introduced by SFQ and EFQ.

with lower ratios, resulting in lower delays for such flows. EFQ by just using the estimates is able to rightly reverse this order. Fig. 7 shows the average misordering delay introduced by the two schedulers plotted with increasing average packet execution times. Note that all the flows have the same reserved processing rates. This plot clearly shows the above conjectured correlation between average misordering delay and average processing time per packet to reserved rate ratio.

D. Simulation Summary

In summary, the simulation shows three main results. One is that the analytically derived worst case misordering delay is almost reached by the SFQ scheduler as shown in Fig. 4(a). Second, EFQ shows a much lower and smoother scheduling delay. This is due to the delay depending on the variance of the processing times rather than the absolute processing times as in SFQ. Third, SFQ introduces unfairness by favoring flows with high processing time to reserved rate ratios. This behavior is not shown by EFQ, which provides fairness over a wide range of processing requirements.

VI. Conclusions

In this work, we have presented an approach to providing QoS guarantees for flows that are processed on nodes in the network. We have shown that network processing applications exhibit very regular and predictable processing patterns, which help overcome the obstacle of theoretically undeterminable computation times of arbitrary programs. The processing time estimations can be approximated by a linear function that we use for admission control. The Estimation-based Fair Queuing (EFQ) algorithm also uses these estimates to fairly and efficiently assign packets to processing engines. The analysis and simulation results show that EFQ performs significantly better in terms of misordering delay and fairness than a SFQ scheduler.

We believe these results are an important step in providing the type of QoS guarantees that are common for bandwidth schedulers in an environment where flows compete for processing resources.

References

[1] Josep M. Blanquer and Banu Ozden, "Fair queuing for aggregated multiple links," in *Proc. of ACM SIGCOMM 2001*, San Diego, CA, Aug. 2001.

[2] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[3] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vincente, and Daniel Villela, "A survey of programmable networks," *Computer Communication Review*, vol. 29, no. 2, pp. 7–23, Apr. 1999.

[4] Dan Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernhard Plattner, "A scalable, high performance active network node," *IEEE Network*, vol. 31, no. 1, pp. 8–19, Jan. 1999.

[5] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner, "Router Plugins - a modular and extensible software framework for modern high performance integrated services routers," in *Proc. of ACM SIGCOMM 98*, Vancouver, BC, Sept. 1998.

[6] Larry Peterson, ed., "NodeOS interface specification," Tech. Rep., AN Node OS Working Group, Jan. 2001.

[7] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott and Karlin, "Scheduling computations on a software-based router," in *Proc. IEEE Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Cambridge, MA, June 2001, IEEE.

[8] Jonathan T. Moore, Michael Hicks, and Scott Nettles, "Practical programmable packets," in *Proc. of the Twentieth IEEE Conference on Computer Communications (INFOCOM)*, Anchorage, AK, Apr. 2001, pp. 49–59.

[9] Hui Zhang, "Service disciplines for guaranteed performance service in packet switching networks," *Proc. of the IEEE*, vol. 83, no. 10, pp. 1374–96, Oct. 1995.

[10] Abhay K. Parekh and Robert G. Gallager, "A generalized processor sharing approach to flow control: The single node case," in *Proc. of IEEE INFOCOM 92*, Florence, Italy, May 1992, pp. 915–924.

[11] Jon Bennett and Hui Zhang, "Worst case fair weighted fair queuing," in *Proc. of IEEE INFOCOM 95*, Boston, MA, Apr. 1995, pp. 120–128.

[12] S. Jamaloddin Golestani, "A self clocked fair queuing scheme for broadband applications," in *Proc. of IEEE INFOCOM 94*, Toronto, Canada, June 1994, pp. 636–646.

[13] Dimitrios Stiliadis and Anujan Varma, "Rate proportional servers: A design methodology for fair queuing algorithms," *IEEE/ACM Trans. on Networking*, vol. 6, no. 2, pp. 164–174, Apr. 1998.

[14] Pawan Goyal, Harrick M. Vin, and Haichen Cheng, "Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks," in *Proc. of ACM SIGCOMM*, Palo Alto, CA, Aug. 1996, ACM, pp. 157–168.

[15] Pawan Goyal, Harrick M. Vin, and Haichen Cheng, "A hierarchial cpu scheduler for multimedia operating systems," in *Proc. of the Second USENIX Symp. on Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996, pp. 107–121.

[16] Jon C. R. Bennett and Hui Zhang, "Hierarchial packet fair queuing algorithms," in *Proc. of ACM SIGCOMM*, Palo Alto, CA, Aug. 1996, ACM, pp. 43–56.

[17] Tilman Wolf and Mark A. Franklin, "CommBench - a telecommunications benchmark for network processors," in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, Apr. 2000, pp. 154–162.

[18] John DeHart, William Richard, Edward Spitznagel, and David Taylor, "The smart port card: An embedded UNIX processor architecure for network management and active networking," unpublished.

[19] Tom Chaney, Andy Fingerhut, Margaret Flucke, and Jonathan Turner, "Design of a gigabit ATM switch," in *Proc. of IEEE INFOCOM 97*, Kobe, Japan, Apr. 1997.

[20] Virginie Galtier, Kevin L. Mills, Yannick Carlinet, Stafan Leigh, and Andrew Rukhin, "Expressing meaningful processing requirements among heterogeneous nodes in an active network," in *Proc. of the Second International Workshop on Software and Performance*, Ottawa, Canada, Sept. 2000.

[21] Sumi Yunsun Choi, Jonathan S. Turner, and Tilman Wolf, "Configuring sessions in programmable networks," in *Proc. of the Twentieth IEEE Conference on Computer Communications (INFOCOM)*, Anchorage, AK, Apr. 2001, pp. 60–66.

[22] ATM Forum Technical Committee, *Private Network-Network Interface Specification Version 1.0*, Mar. 1996.