Predictive scheduling of network processors

Tilman Wolf [a,*], Prashanth Pappu [b], Mark A. Franklin [b]

[a] *Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003, USA*
[b] *Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA*

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 (NPs) 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 service 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, which can be exploited for scheduling purposes. We present two such predictive processor scheduling algorithms that aim at providing service guarantees as well as improving the performance of the NP by increasing the instruction data locality. Simulation results show that these algorithms provide significantly better performance than processor scheduling algorithms that do not take packet processing times into consideration.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling; Network processors; Programmable networks; Fair queuing; Locality-aware scheduling

1. 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 paths of routers that, for performance reasons, are usually implemented in custom logic. To overcome this obstacle, it has been proposed placing general-purpose processing engines in the data path of routers. Such network processors (NPs) extend the traditional store-and-forward paradigm to store-process-and-forward, an approach that opens vast possibilities for novel networking applications.

Scheduling the use of the multiple processing engines on a given NP is an important element in determining system performance. Through scheduling it is possible to provide fair sharing of the processing resources between competing flows and to provide delay bounds for packets traversing the system. This is crucial since an important goal in networking is the development of network infrastructures that provide quality-of-service (QoS) guarantees.

---

* Corresponding author.
E-mail addresses: wolf@ecs.umass.edu (T. Wolf), prashant@arl.wustl.edu (P. Pappu), jbf@ccrc.wustl.edu (M.A. Franklin).

A second important aspect of scheduling is its impact on the performance of the processing system. Typically, the size of on-chip caches of NPs is very small and instructions for only a single program can be stored. If, in response to the needs of an incoming packet, the scheduler assigns a different program to a processing engine a large number of cache misses are triggered because the new instructions have to be swapped into cache. Our measurements show that this "cold cache" behavior can reduce the overall processing performance by as much as 25%. Developing a scheduler that is locality-aware and attempts to reuse currently cached instructions results in lowering the number of cold cache misses that occur.

A large number of scheduling schemes have been proposed for a number of domains in computer science. In particular processor scheduling in operating systems and link scheduling on routers have been studied extensively. However, due to NP system characteristics, neither type of scheduling is applicable to NPs. These characteristics include:

1. For most packets, the cost of context switching exceeds the processing time.
2. The processor execution time associated with a packet arrival is generally unknown since, in the most general case, it is a function of the type of processing to be performed, the packet contents, and the state of the processor's cache memory.

The first point indicates that traditional CPU scheduling schemes are not applicable to NPs since they are based on the ability to switch between different processes to ensure fairness. The second point limits the applicability of link scheduling algorithms for this domain. In link scheduling, it is assumed that it can be known beforehand how long a packet occupies the resource. For the case of a link this depends on the packet size and the speed of the link. But for the packet processing associated with NPs, this is not known in advance since the state of the processor's cache is generally unknown. Thus the guarantees on fairness and delay bounds associated with link oriented scheduling algorithms do not apply. This lack of de-

terministic resource usage makes the scheduling problem for NPs particularly hard. This also impacts how explicit or implicit admission control and reservations can be done in such an environment.

Our approach to developing a useful scheduling algorithm in the NP domain uses *processing time predictions*. We present measurement results that show that typical processing on a NP is regular and the execution time is highly predictable. Thus, the time that a packet occupies a processing engine can be estimated and used for the scheduling decision. Once the processing is completed, the actual processing time is fed back to the scheduler for accurate accounting. With this approach long-term fairness and delay bounds can be ensured, even if occasionally a predicted processing time is inaccurate.

We present two scheduling algorithms, which address the three key challenges in scheduler design. We would like to achieve:

1. Fairness between independent flows.
2. Bounds on packet delay.
3. High system performance.

Estimation-based fair queuing (EFQ) is a scheduling scheme that addresses the first point. Using reservations and admission control, the requirements of each flow are known and the scheduler can enforce fair sharing. In this regime, the NP is typically not overloaded because admission control is in place. As a result, delay bounds on conforming flows can be given. A different situation can occur when the NP is used in a best-effort network, like the current Internet. For this regime, we propose another algorithm, locality-aware predictive (LAP) scheduling, which reduces cold cache misses and results in higher processing rates. For each algorithm, we present simulation results and compare the performance to other algorithms that do not use processing time estimations. The results show that EFQ and LAP can achieve the desired properties of fairness, delay bounds, and high system performance.

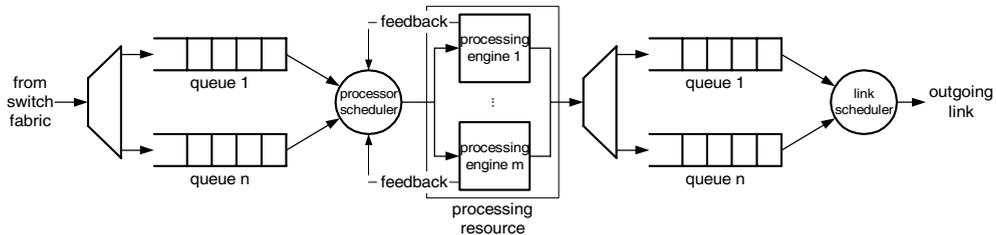Section 2 formalizes the scheduling problem and describes the desired properties of a scheduling

Fig. 1. Scheduler system outline.

algorithm. Section 3 shows measurement results on which the processing time predictions are based. Section 4 introduces EFQ and shows a proof of the fairness and delay bounds as well as simulation results. Section 5 discusses the LAP scheduling algorithm and simulation results that show the performance gains from instruction data reuse. Section 6 describes how EFQ and LAP can be combined in a single system. Related work is discussed in Sections 7 and 8 summarizes and concludes this paper.

2. Scheduling problem

A router that is equipped with a NP has two types of resources for which scheduling is necessary. One is the link (or the interface to the switching fabric on the input port), the other is the processing resource on the NP. The link scheduling issues have been studied extensively. A less investigated area is the scheduling of the processors in such a system. Since we consider a system where arbitrary code may be processed by the NP, we assume that it is not possible to statically schedule the processing engines (e.g., by limiting the execution time for each packet to less than the interarrival time of packets). This also limits the ability to combine the processor scheduling and the link scheduling into a single scheduler, since packet processing time does not necessarily correlate with packet size. As a result, it is necessary to queue packets and then assign them to processors as they become available. It is this scheduling decision where QoS, fairness and efficient processor operation can be ensured.

The data flow through the NP is illustrated in Fig. 1. Packets that are queued for processing need to be assigned to processors when these become available. The processor scheduler can choose any one packet from the n queues and assign it to any of the m processing engines if they are idle. After processing, packets are again queued in per-flow queues before the link scheduler assigns them to be transmitted on the link. This system abstraction captures the essence of most current NP architectures, where multiple simple processor engines handle packet processing in parallel [15,16]. To maintain generality, we do not consider interactions between processing engines (e.g., contention for memory access) as they are specific to particular NP designs.

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 [3]. In either case, the essential problem reduces to designing an efficient scheduling algorithm for sharing a single processing resource.

The scheduling algorithms can address different goals. For our purpose, we are interested in three key points:

1. *Fair sharing of resources:* The scheduler should ensure that flows get access to processors evenly. That is the processor scheduler should ensure that no flow exceeds its fair share of processor usage.

2. *Low delay:* In order to minimize the effect of processing on the data flow, the scheduler should also aim at reducing the overall delay that a packet experiences. This also implies that the delay variation (i.e., jitter) should be minimized.

3. *Good performance:* The scheduler should be work-conserving. That is if a processor is idle and a packet is available for the processor, the scheduler should not keep the processor idle. In addition, the scheduler should avoid "cold" instruction caches, which reduce the performance of the processing system. This effect is explained in more detail below.

These goals are almost identical for scheduling in the link bandwidth domain. However, there is one key reason, why link oriented scheduling algorithms cannot be simply used for processor scheduling: In theory, processing time of an arbitrary piece of instruction code on a general-purpose processor cannot be determined beforehand (because it is a version of the halting problem for turing machines). Most bandwidth scheduling algorithms rely on knowledge of packet sizes (which corresponds to transmission times on the link resource). Another reason is that transmissions of packets of the same size always take the same time. However in processor scheduling, the processing time of a packet depends on the packet data and the state of the processor as it was left by the previous packet (i.e., the state of the on-chip cache). Therefore it is necessary to consider new scheduling algorithms that take these issues into account.

Despite the theoretically arbitrary processing times for packets, the processing characteristics in real systems show high levels of regularity, which can be exploited for scheduling.

3. Processing characteristics

The scheduling algorithms presented in this paper are based on the assumption of predictable packet processing times. This section presents some measurement results of processing characteristics in the NP environment.

3.1. Predictability of processing times

The nature of packet processing causes the applications to repeatedly execute the same code over the packets that are passed through the processor. This leads to good predictability of processing times as the following results show.

3.1.1. Measurements

The processing time of packets depends on a variety of factors: instruction code ("application"), packet data, processor state, processor configuration, etc. To illustrate the impact of application, packet data, and packet size, we performed measurements of packet processing times. Four applications were considered: encryption, compression, forward error correction (FEC), and IP forwarding. The first three are in the category of "payload-processing applications" [27]. IP forwarding is a "header-processing application". For the measurements, the Washington University Gigabit Switch [4] enhanced with the single-processor linecard [9] was used. The processor engine in this system is a single 167 MHz Pentium. The software environment for the processing utilized the crossbow/active network node operating system [7,8]. The static object code size for the applications ranges from 4 kB (IP forwarding) to 34 kB (CAST encryption). Several thousand packets were sent through the programmable router and the overall processing time (from interface to interface) for each packet measured. The interarrival time of packets was large enough to not cause queuing delays. This process was repeated for different packet sizes and applications.

Fig. 2 shows the processing time for packets of different sizes using the three applications. The error bars indicate the 95% percentile of processing time. For encryption and FEC, the processing times are very close to the average. For compression, which is a data dependent computation, the variations are slightly higher. Note that we use time as the metric 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 partic-
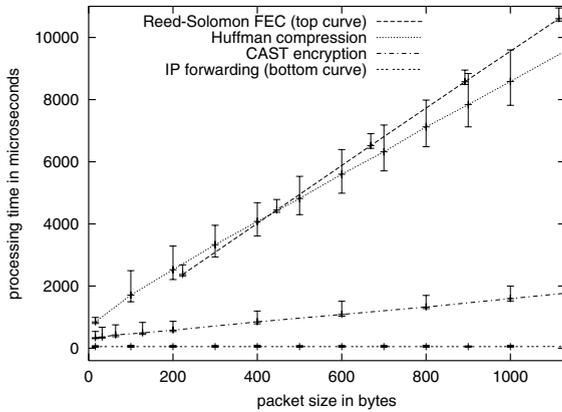
Fig. 2. Packet processing times for programmable router applications. The error bars indicate the 95% percentile of processing times.

ular router system, where the packets get processed, as described in [11].

3.1.2. Processing time approximation

For IP forwarding, the processing time is practically constant for all packet sizes, 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 define the estimated processing time t_e of a packet of length l when processed by application a as

$$t_e(a, l) = \alpha_a + \beta_a l, \tag{1}$$

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. Using the measurements discussed in the previous section, the average values for these appli-

cation parameters are given in Table 1. To simplify notation, $t_e(p)$ be the estimated processing time of packet p, which is of length l and uses application a.

3.1.3. Online estimation

The parameters α_a and β_a in Table 1 have been determined from traces. But it is also possible to determine these parameters online and improve them using simple linear least squares regression techniques. As packets are processed the scheduler can maintain variables denoting the sums, $\sum t_{e,i}$, $\sum l_i$, $\sum t_{e,i}^2$, $\sum l_i^2$, $\sum(t_{e,i}l_i)$ for each application a. These variables are updated on the arrival of a new (c_{n+1}, l_{n+1}) pair and on completion of processing of a packet. The parameters to be used in the estimation can then be computed as regression coefficients:

$$\beta_a = \frac{\sum_n t_{e,i}l_i - \sum_n t_{e,i} \sum_n l_i/n}{\sum_n l_i^2 - \sum_n l_i \sum_n l_i/n}, \tag{2}$$

$$\alpha_a = \sum_n t_{e,i} - \beta_a \sum_n l_i. \tag{3}$$

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

Table 1
Packet processing parameters

Application a	Per-packet cost α_a (µs per packet)	Per-byte cost β_a (µs per byte)	Cold cache penalty π_a (µs per packet)	Expansion factor γ_a
IP forwarding	51	0	70	1
Encryption	320	1.3	170	1
Compression	970	7.6	950	0.13–0.34
FEC coding	320	9.2	175	1.14

always processed together. Even if the processing time cannot be estimated accurately at all, the proposed scheduling algorithms still perform correctly (e.g., EFQ still guarantees fairness) albeit at a cost of lower performance (e.g., increased packet delay).

3.2. Cold cache penalty

Typical NPs are equipped with small on-chip caches (16–32 kB) due to die size limitations. These caches can only hold data for the most recently executed program. Data in the instruction cache can be reused by the processor if subsequent packets require the same program. Data cache information containing packet-dependent data can less easily be reused, since it changes with every packet. Changing the program that a NP executes, causes the caches to become cold, which results in an execution time penalty associated with the initial loading of the cache with new application instructions. This can have a significant negative effect on overall NP performance.

With the same measurement setup as above, this effect of "cold caches" can also be shown and quantified. These measurements are shown for the encryption application in Fig. 3. When sending a stream of packets, which require the same application, the first packet encounters a cold cache. For subsequent packets, the processing time is reduced due to locality in the instruction code and

Fig. 3. Cold cache effects on packet processing times. The results are shown for the encryption application.

the resulting warm cache. The measurements indicate that the cold cache penalty is independent of the packet size. Table 1 shows the average cold cache penalty, π_a, for all applications.

The LAP scheduler discussed in Section 5 aims at minimizing this cold cache penalty by assigning packets to processors that just completed processing the same application as required by the packet.

3.3. Reservations

A key component of QoS 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.

3.3.1. 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. For a single packet, γ_a is defined as the size of the packet after processing divided by the size of the packet before processing. This factor is also shown in Table 1. Note that the expansion factor can be dependent on packet size and data as for the compression application.

3.3.2. 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.

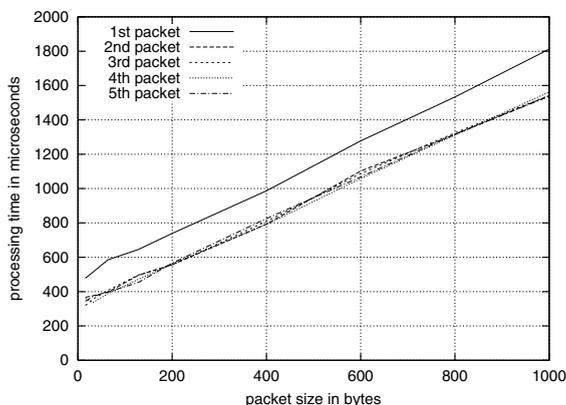A reservation for a flow j with incoming bandwidth B_j that is processed by application a

needs to reserve $\gamma_a B_j$ 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. The fraction B_j/l_j is the average number of packets per second that a router needs to process. For each, a processing time of $t_e(a, l)$ (see Eq. (1)) is necessary:

$$P_j = \frac{B_j}{l_j} t_e(a, l) = \frac{B_j}{l_j} (\alpha_a + \beta_a l). \qquad (4)$$

Thus, flow j can be admitted to any router that has P_j processing power and $\gamma_a B_j$ outgoing bandwidth available.

3.3.3. Processing location

When using reservations, it is necessary to determine the path of the flow and the location(s), where processing should happen. Ideally, the allocation of resources should be optimal (e.g., best performance or lowest cost). Determining the best path in a traditional network can easily be done (e.g., shortest path on delay metric). However, with additional processing steps in programmable networks, it is necessary to develop a new approach. By combining the transmission and processing cost in a single metric and by modifying the network graph data structure to consider processing, we have shown that an optimal route can be computed efficiently [6]. The details of this work are beyond the scope of this paper.

The estimations for processing requirements for packets can be used in a scheduling algorithm that enforces the fair sharing of processing resources according to each flows reservation.

4. Estimation-based fair queuing

For EFQ, it is assumed that the NP operates in a regime, where admission control is performed, flows reserve bandwidth and processing, and the routers are operated below maximum load. EFQ is built upon the class of rate-proportional servers (RPS) having desirable properties that permit processing time estimates to drive the processor scheduling algorithm.

4.1. Scheduling algorithm

The EFQ scheduler is a mechanism 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:

- 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.
- 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.

In order to do this, we base EFQ on the design methodology of RPS.

4.1.1. Rate proportional servers
4.1.1.1. Definition. RPS are a class of scheduling algorithms designed according to the methodology presented in [24] that allow the designer to trade fairness of the algorithm with implementation complexity. Generally speaking, a RPS is a work-conserving server with the following properties:

- The server has an associated system potential that is updated to reflect the total work done by the server.
- 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 [24] 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 RPS have delay and fairness properties comparable to generalized processor sharing

(GPS). WF^{2}Q+ [2] 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.

4.1.1.2. Packet selection policy. A scheduling algorithm with optimal fairness would have to schedule single processing cycles according to the fluid RPS. However, in NPs, 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 well 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 B and C, in the order in which they are more likely to finish processing. That is, if F_b^j and F_c^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

$$P[F_b^j \geqslant F_c^k] \geqslant 0.5. \qquad (5)$$

That is, there is a greater probability of j finishing before k. Hence, it is the knowledge of the distributions of F_b^j and F_c^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_b^j is

$$F_b^j = P_b + \frac{W_b^j}{R_b}, \qquad (6)$$

where P_b is the potential and R_b is the rate of service reserved by flow B. While these are known in advance when determining F_b^j, W_b^j, which represents the service time required by packet j, is not. Thus, the random variable F_b^j is directly determined by W_b^j.

Start-time fair queuing (SFQ) [14] (with a modified system virtual time) and WF^{2}Q+ [2] are scheduling algorithms belonging to this class that represent the extremes with respect to the amount of knowledge of F_b^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_b, 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.

4.1.1.3. 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_b^j. To quantify these delays, assume that a scheduler of this class can be characterized by random variables χ_{b^j,c^k}, which denote the event that the scheduler (with its knowledge of W_b^j and W_c^k) makes a mistake in ordering packets j and k. That is, $P[\chi_{b^j,c^k} = 0]$ is the probability that the scheduler orders the packets of these two flows correctly, while $P[\chi_{b^j,c^k} = 1]$ is the probability that the scheduler makes a mistake in the ordering. Then, the average misordering delay, δ_b, as seen by a packet of flow B is the additional delay caused by the scheduler misordering packets of flow B and flow C, which is

$$\delta_b = P[\chi_{b^j,c^k} = 1] \frac{R_c}{R} \left(P_c + \frac{W_c^j}{R_c} - P_b - \frac{W_b^i}{R_b} \right). \qquad (7)$$

This accounts for the time spent by the server in servicing additional traffic from flow C before

processing packets from flow *B*. 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 3.1, which improves the schedulers knowledge of W_b^j.

4.1.2. 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 RPS can be changed to use these estimates, EFQ is derived by modifying WF^2Q+ which is known to have the tightest delay bounds and low time-complexity among bandwidth schedulers. EFQ, like WF^2Q+, uses a notion of system virtual time (system potential), defined by

$$V(t + \tau) = \max\left(V(t) + \tau, \min_{i \in B(t+\tau)} S_i\right), \qquad (8)$$

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^2Q+ a RPS. 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 [2].

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

$$S_i = \max(F_i, V(a_i^k)), \qquad (9)$$

if queue *i* is empty, else

$$S_i = F_i. \qquad (10)$$

EF_i is updated using

$$EF_i = S_i + \frac{t_e(p_i^k)}{r_i}, \qquad (11)$$

where $t_e(p_i^k)$ is the estimated number of instructions required to process packet *k* (see Eq. (1)). When the processor finishes processing this packet, the actual finish tag F_i is updated using feedback from the processor:

$$F_i = S_i + \frac{A_i^k}{r_i}, \qquad (12)$$

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.

4.1.3. Example

The following illustrates the behavior of EFQ and compares it to that of SFQ and WF^2Q+. 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. 4 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. 4 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^2Q+ 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
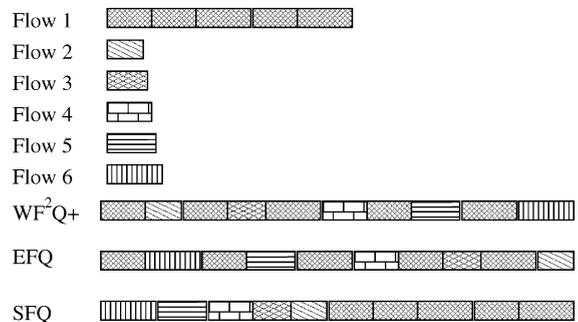

Fig. 4. EFQ scheduling example. The upper half shows the input to the scheduler, the lower half the scheduling sequence for different scheduling algorithms. All packets are of the same length and are processed by the same application. The figure shows the actual execution times of packets as their size and the processing order derived by different scheduling disciplines.

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. 4. 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.

4.1.4. Analysis

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

$$\delta_{\text{SFQ}} = \sum_{i=1}^{N} \frac{A_i^{\max}}{R} - \frac{A_a^{\max}}{R_a}. \tag{13}$$

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 Eq. (7), since the scheduler

can make a mistake only when $P_b \leqslant P_a$. Results below also show that SFQ actually favors (i.e., gives lesser delays to) flows with packets which require greater average normalized service (i.e., higher $t_e^{\text{avg}}(p_a)/R_a$).

To analyze EFQ, assume that for a given packet length, the packet execution time estimates obtained in Section 3.1 can be represented by uniform random variables W_a^j lying in the range $[t_e(p_a^j) - V_a^j, t_e(p_a^j) + V_a^j]$. The EFQ scheduler misorders packet j and k when it determines that

$$P_a + \frac{t_e(p_a^j)}{R_a} \geqslant P_b + \frac{t_e(p_b^k)}{R_b}, \tag{14}$$

but the actual processing times are such that

$$P_a + \frac{A_a^j}{R_a} \leqslant P_b + \frac{A_b^k}{R_b}. \tag{15}$$

In the worst case, we get

$$\frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} \leqslant P_b - P_a \leqslant \frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} + \frac{V_a^{\max}}{R_a} + \frac{V_b^{\max}}{R_b}. \tag{16}$$

Hence from Eq. (7), the misordering delay for packet j due to packet k is limited to

$$\delta_a = P[\chi_{a^j,b^k} = 1] \frac{R_b}{R} \left(\frac{V_b^{\max}}{R_b} + \frac{V_a^{\max}}{R_a} \right) \tag{17}$$

and the worst case misordering delay is bounded by

$$\delta_{\text{EFQ}} = \sum_{i=1}^{N-1} \frac{V_i^{\max}}{R} - \frac{V_a^{\max}}{R} + \frac{V_a^{\max}}{R_a}. \tag{18}$$

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+.

4.2. Evaluation

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

4.2.1. Simulation setup

To compare the delay characteristics of the two schedulers, we use the following simulation setup. First, we use the traces of 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 2 GHz (about 10 times the speed of the processor on the smart port card [9] 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 processing rate and adjust their sending rates to just saturate their share of the processing resource. It is assumed that no queuing delays occur, so the simulation results only consider scheduling delays.

4.2.2. Packet delay

Fig. 5 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 μs, 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. 5(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 μs. The maximum delay actually observed in Fig. 5(a) is about 6100 μs, implying an observed maximum misordering delay of $6100 - 163 = 5937$ μs.

For EFQ, much lower delays can be seen in Fig. 5(b). This illustrates two things. First, given the small execution time of forwarding as compared to other applications, the finish times of the packets of this flow were 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., Eq. (16) was not satisfied for most comparisons of finish times). Second, the worst case delay that could be experienced by these packets is only 1312 μs 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 $900 - 163 = 737$ μs, which is about one order of magnitude smaller than for SFQ.

Fig. 6 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. 6(a) are about three times the interarrival time of the packets. Fig. 7 shows the delays experienced by a flow being processed by the FEC application which requires much greater processing time per
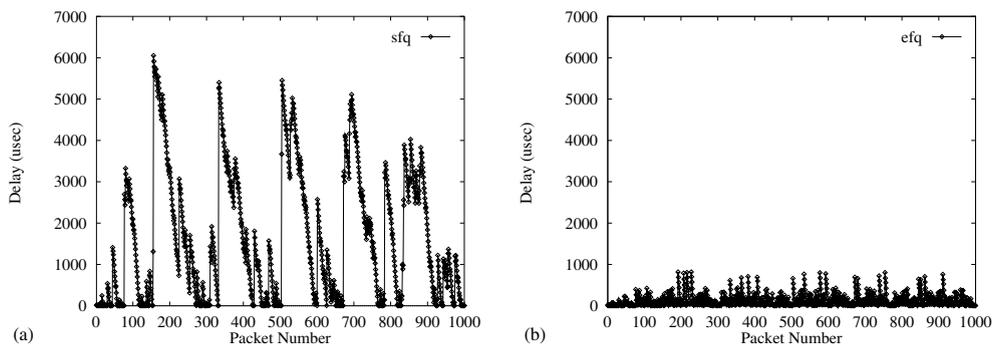

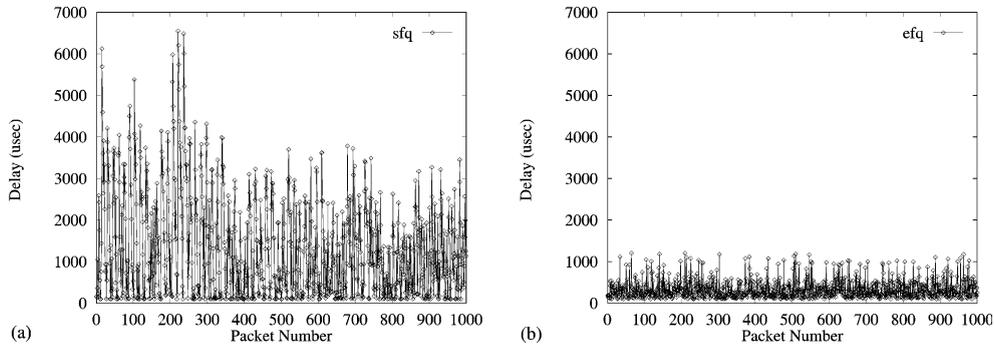Fig. 5. Packet delays for a flow processed by IP forwarding: (a) SFQ and (b) EFQ.

Fig. 6. Packet delays for a flow processed by CAST encryption: (a) SFQ and (b) EFQ.
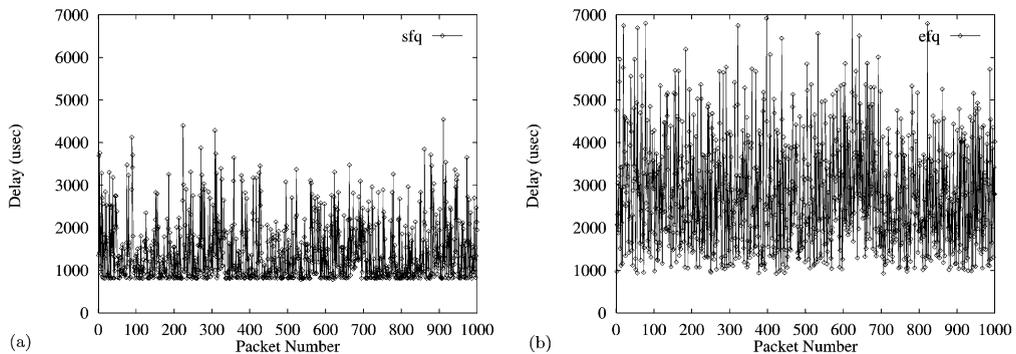

Fig. 7. Packet delays for a flow processed by Reed–Solomon FEC: (a) SFQ and (b) EFQ.

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. This indicates an average negative misordering delay, while delays due to EFQ are just about the interarrival time of the packets. Two important conclusions can be drawn from these plots:

- SFQ gives much higher misordering delay bounds than EFQ.
- Across flows, while the misordering delays due to EFQ are on an average close to zero, they vary from high positive 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.

The second point indicates a bias of SFQ.

4.2.3. Biased delay bounds due to SFQ

The bias of SFQ 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., $t_e^{avg}(p_a)/R_a$).

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 with lower
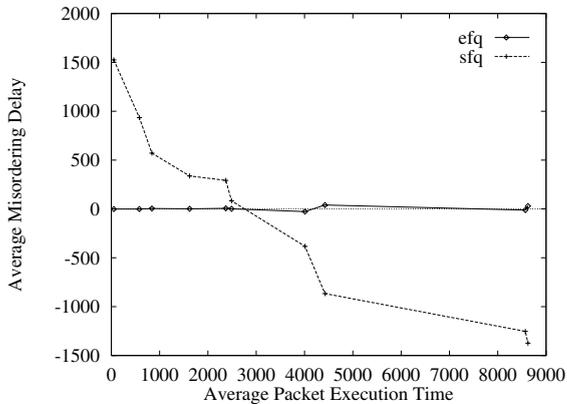
Fig. 8. Variation in minimum packet delay for different flows introduced by SFQ and EFQ.

ratios, resulting in lower delays for such flows. EFQ by just using the estimates is able to rightly reverse this order. Fig. 8 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.

4.2.4. 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. 5(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.

Despite the favorable properties of EFQ, it might not be possible to use it in all networks because it requires flow reservations. In a best-effort network, like the current Internet, the requirements of flows are not explicitly expressed. For such an environment, it is desirable to

schedule packets in a way to maximize overall system performance and keep up with best-effort services.

5. Locality-aware predictive scheduling

The LAP scheduling algorithm aims at reducing the cache miss rates that are due to cold caches. For this purpose, the scheduler estimates the processing time of all packets that are queued in the system at the time. Based on the estimates, the scheduler partitions the set of processing engines of the NP into application groups. Packets are then sent to processors that belong to their respective application group. This requires that the scheduler distinguish between the different processing engines and not consider them as one processing resource (as it is done with EFQ).

5.1. Scheduling algorithm

The execution time of a packet depends on the state of the cache of the processor when it is processed. A cache is said to be cold if the application required by a newly assigned packet differs from the application just completed. If the cache is warm (i.e., not cold), the processing time is $t_e(a, p)$. If the cache is cold, a penalty of π_a is added to the processing time $t_e(a, p)$. The LAP scheduling algorithm considers this. To compare LAP's performance, we also define throughput-optimal (T-Opt), which is optimal in terms of least cold caches, and first-come-first-serve (FCFS), which is optimal in terms of least delay variation (as defined below).

5.1.1. Locality-aware predictive

LAP tries to group processors such that each group processes one application and thus keeps a warm cache for this application. The size of each group is determined by the amount of processing pending for packets in queue memory. We define Q_t as the set of packets in queue memory at time t of which the LAP scheduler is aware. For each application a, LAP computes the amount of processing time needed by all

packets of this application in Q_t. This is expressed as the fraction of processing time for application a over the total processing time of all packets in Q_t:

$$f_{a,t} = \frac{\sum_{p \in Q | \text{app}(p)=a} t_e(a,p)}{\sum_{p \in Q} t_e(a,p)}, \tag{19}$$

where $\text{app}(p)$ is the application used by packet p. According to these fractions, $f_{a,t}$, the set of processing engines on the NP are partitioned into application groups. If n processing engines are available in total, LAP will assign $f_{a,t}n$ (rounded to an integer value) of them to process application a. This way the proportions of processing engines matches the processing estimates and the number of cold caches is minimized. The assignment of processors to applications changes dynamically to match the required processing times of packets in Q_t as LAP recomputes the partition for each scheduling decision.

The effectiveness of LAP is based on the assumption that the processing time for packets is predictable from their size and the application they execute. LAP performance also depends on the number of packets that the scheduler is of and that are available in queue memory. We define this number as $|Q_t| = q$. In the simulation results below, we can see that LAP performance increases with more packets in queue memory (larger q) as LAP can choose from a larger set of packets.

For comparison purposes we define T-Opt as the algorithm that achieves maximum locality (and thus maximum throughput) by being allowed to pick any packet out of the entire packet stream (independent of Q_t). T-Opt executes all packets of one application before it switches the processor to another. Thus, the only cold caches are due to compulsory cache misses for the first packet of an application. This strategy, though not realistic for actual implementation, gives an upper bound on the possible performance.

A naive instance of best-effort scheduling is FCFS. In this scheme, packets are assigned to processors in the order of their arrival. If a processor u becomes available at time t, the oldest packet in queue memory Q_t is sent to u: The

schedule does not take any locality into account. It is optimal in terms of variation in delay for packets since it does not re-order packets and keeps the delay for each packet in a given flow the same.

5.2. Evaluation

In order to evaluate the performance of LAP several performance metrics need to be defined.

5.2.1. Performance metrics

The performance of a scheduler can be defined in several (sometimes conflicting) ways. The performance depends in large part on the order of packet execution and the resulting processing time for the packet set. We define the following performance criteria:

- *Throughput:* The throughput is defined as the amount of data that is processed in a given amount of time. This is the key performance parameter, since generally NPs are aimed at processing as much data as possible.
- *Fraction of cold caches:* The fraction of cold caches is the number of times a packet p is assigned to a processor with a cold cache divided by the total number of scheduled packets. This fraction is an indicator of how much locality awareness a scheduling scheme shows. The lower the fraction, the fewer cold cache penalties are incurred.
- *Delay variation:* The delay variation is a metric similar to the misordering delay in EFQ, but it is measured in terms of the order of packets instead of actual delay. The delay variation is the standard deviation of the variation in the order of packets being processed as compared to their arrival order. The larger the delay variation, the more misordering occurs. While it is necessary to change the order of packet processing to make use of locality in reducing the negative effects from cold caches, the goal is to keep the delay variation low.

Using these performance measurements, LAP scheduling can be compared to naive and optimal scheduling.

5.2.2. Simulation

The evaluation of the scheduling algorithms is done using a trace-driven simulation. Packet traces were obtained from the packet processing time measurements described above. Only the three payload processing applications, FEC, encryption, and compression, are considered. Traces of 100,000 packets are generated having an equal share of bandwidth for each application. To simulate more than three applications, the original traces are replicated with different application identifiers. We assume that a processor can only have one application in its instruction cache at any

time, which is reasonable for the small cache sizes considered. These traces are used as input to a discrete event simulator that emulated the behavior of the scheduler and the processors. Simulations are performed over a variety of configurations.

5.2.3. Basic operation and adaptation to workload changes

To illustrate the basic operation of each of the algorithms, we look at the case where we have three applications, 16 processors, and 64 packets in queue memory. The application workload is
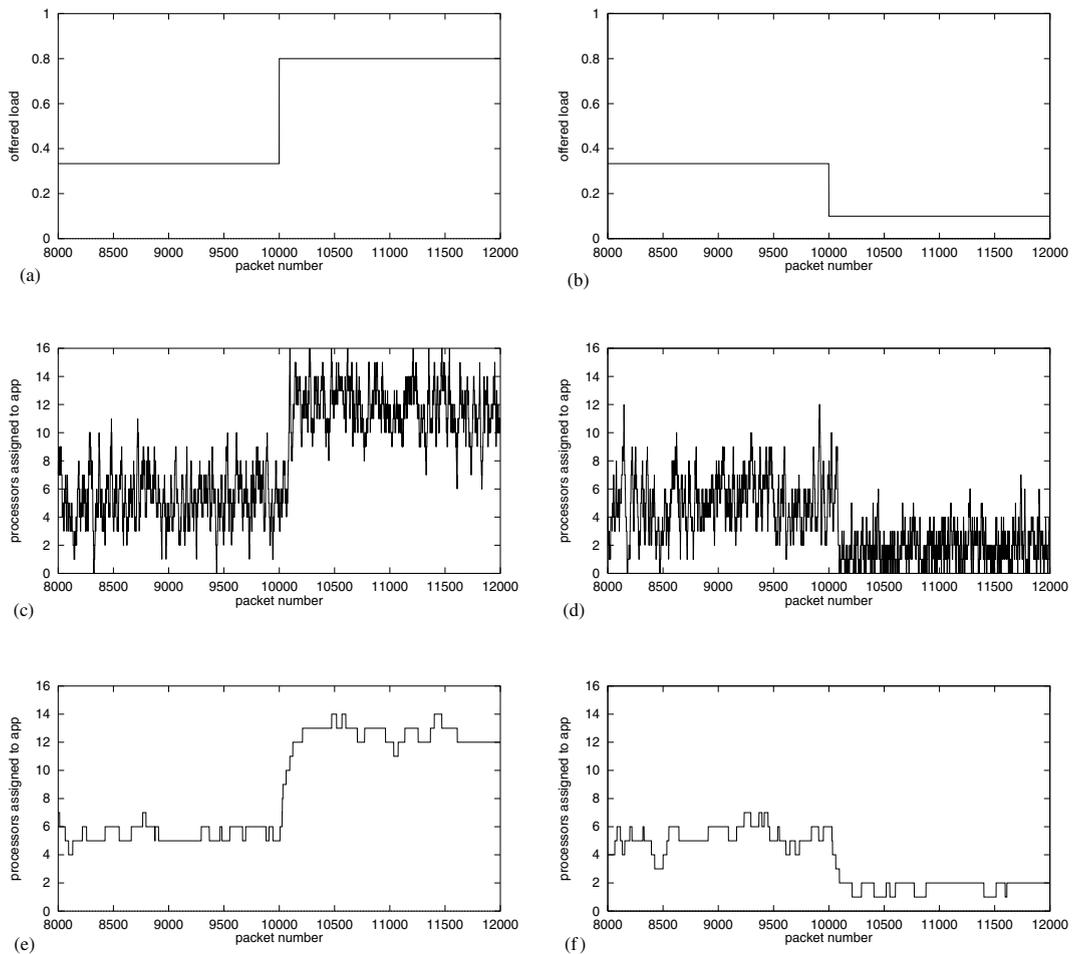

Fig. 9. Processor assignment comparison between FCFS and LAP. The scheduling assignments for applications 2 and 3 are similar and only one set is shown: (a) offered load (app 1), (b) offered load (app 2/3), (c) FCFS (app 1), (d) FCFS (app 2/3), (e) LAP (app 1) and (f) LAP (app 2/3).

such that the first 10,000 packets require equal processing. Thus, each application on average should be processed on one third of the processors. After 10,000 packets, the workload changes, such that application 1 requires 80% of the processing and applications 2 and 3 require 10% each (see Fig. 9(a) and (b)). This is used to illustrate the adaptability of the various algorithms to changes in the workload. Fig. 9(c)–(f) show the FCFS and LAP scheduling algorithms. The lines show how many processors have warm caches for each application (i.e., how many processors process each application at that moment) for packets 8000 through 12,000.

Each change in the number of assigned processors (*y*-axis) causes a cold cache, which reduces the overall performance. FCFS scheduling shows the expected "random" behavior. Since packets are scheduled in the order of arrival, no locality is explicitly exploited and the number of processors executing a given application changes frequently. This behavior leads to a large number of cold caches and low performance. A smooth scheduling behavior is produced by LAP scheduling, because it partitions the processors according to the processing requirements. Fig. 9(e) and (f) show that the partitioning follows very closely to the offered load as shown in Fig. 9(a) and (b).

Also, LAP adapts quickly to changes in the workload. LAP reaches a processor assignment that corresponds to the offered load within a few hundred packets of the change in workload (3–4 times the number of packets in queue memory). During this period, packets from before the change are still in queue memory and influence the scheduling decision.

5.2.4. Throughput

Fig. 10 shows a throughput comparison of LAP with FCFS and T-Opt. The number of processors considered is 16 and the number of applications is 30. Since LAP depends on the number of packets in queue memory, this value is varied on the *x*-axis. FCFS has the lowest throughput of about 85% of T-Opt. This can be expected, since FCFS does not take locality into account. For a very small number of available packets, LAP is close to FCFS, since the number of packets from which the algorithm can select is small and locality can only be
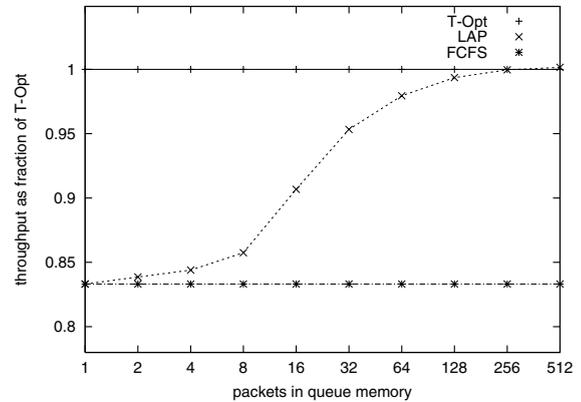

Fig. 10. Throughput for LAP compared to FCFS and T-Opt. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

maintained for short times. With about 16–64 packets, LAP performs significantly better than FCFS. For large numbers of packets, LAP converges towards the throughput of T-Opt.

5.2.5. Cold cache fraction

To illustrate the correlation between the use of locality information and throughput, Fig. 11 shows the cold cache fraction of packets for the same parameters as used in Fig. 10. The cold cache fraction gives the percentage of packets that are executed with a cold cache (i.e., do not make use of locality). FCFS has the highest rate of cold caches with about 96%. This is due to the random as-
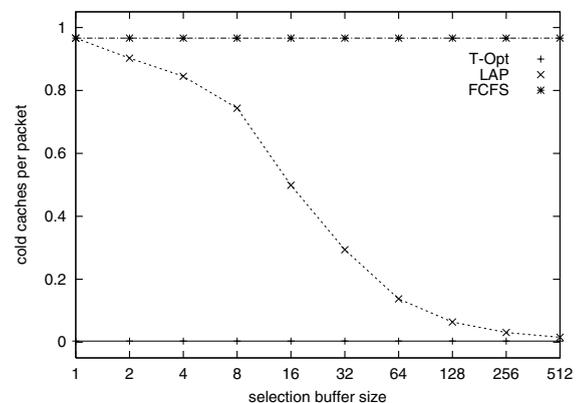

Fig. 11. Cold cache fraction for LAP compared to FCFS and T-Opt. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

signment of packets to processors in FCFS, which causes only 1 in 30 assignments to be to a processor with warm caches (because there are 30 applications).

The cold cache fraction for LAP shows a trend that corresponds to the throughput performance shown in Fig. 10. For small numbers of available packets, the number of cold caches is close to that of FCFS. As more packets are available, cold caches drop and LAP converges towards T-Opt.

5.2.6. Delay variation

Fig. 12 shows the standard deviation of the variation in packet order for FCFS and LAP. The delay variation for T-Opt is arbitrarily large and thus not plotted here. For FCFS, there is no variation, because packets maintain their order. One can see that LAP shows increasing delay variation for increasing numbers of packets in queue memory. This is expected since the reordering is roughly limited to the number of available packets. The large variation of delay for increasing numbers of packets indicates that there is a tradeoff between achieving more cache locality and delay variation.

In contrast to EFQ, this delay is considerable. For EFQ, the overall misordering delay (see Fig. 8) was on average close to zero. For LAP, this can not be achieved anymore, as the scheduler aims at higher system performance (through fewer

Fig. 12. Delay variation for LAP compared to FCFS. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

cold caches) at the cost of higher packet delay variations.

5.2.7. LAP complexity

Finally, the usefulness of these scheduling algorithms depends on how efficiently they can be implemented in hardware. LAP has constant processing cost per packet, making it well suited for high performance systems. The following briefly discusses a possible data structure for LAP that can be implemented in hardware and has $O(1)$ update complexity.

There are four components necessary for LAP scheduling: the current values of the processing time estimation for each application, the application for which each processing engines has a warm cache, and a list of packets pending processing for each application in order of packet age, and a list of all packets in order of packet age. Each of these structures can be updated in constant time when a packet is received or scheduled. The update of the processing time requirements can be done every time a packet is entered into the packet buffer by adding its expected processing time. When a packet is removed, the processing time is subtracted. Similarly, the warm caches can be adjusted by incrementing and decrementing as processors change the applications that they process. An update occurs only when a packet enters or leaves the buffer. Thus, the complexity is $O(1)$ per packet. Maintaining lists of packets for different applications that are sorted by the age of the packets can also be done in constant time. Since the age of packets corresponds to the arrival order, a simple queue can be used. Updates to queues can be done in $O(1)$ time per update. There has been much work done in implementing efficient queuing systems of this sort [5].

5.2.8. Evaluation summary

The evaluation indicates that good throughput performance can be achieved if 32 or more packets are available to chose from. However, the delay variation increases significantly causing large-scale re-ordering of packets in the data stream. Therefore, the throughput and delay seem to have a "sweet spot" around 16–32 available packets. To operate the system in this region, the number of packets to chose from could be limited artificially
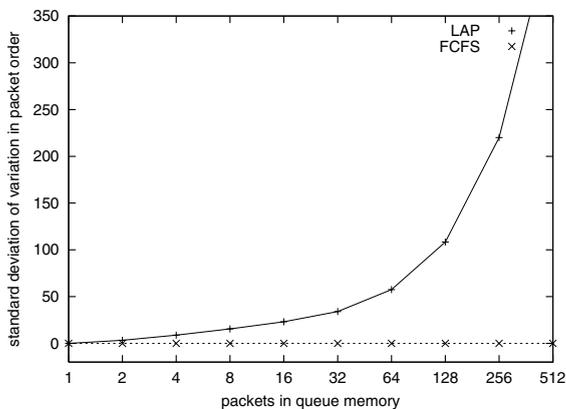
to 16 or 32. If more packets are available, the scheduler can only chose from the 16 or 32 oldest packets in the queue.

Another critical observation is that the LAP algorithm causes the system to perform better as it becomes more loaded. For small numbers of packets (i.e., low load), the scheduler might cause a few cold caches. For more packets, the number of cold caches approximates the optimum.

In summary, LAP is a good scheduling algorithm for a system that is backlogged. LAP improves the throughput of the system by avoiding cold caches. One main drawback is the delay that it might impose on packets.

6. Combination of LAP and EFQ

The EFQ and LAP scheduling algorithms address two different problems in NP task scheduling. EFQ ensures fair sharing of resources and bounds the misordering delay of packets. LAP increases the throughput of the system as it makes use of instruction locality. Both schedulers also operate in different usage regimes of a programmable router. LAP performs best when packets are backlogged in queue memory, which is characteristic for a system, where the processing resource is a bottleneck. EFQ ensures fair sharing and delay bounds when the processing resource is not oversubscribed. It is also necessary that flows adhere to the specified data rate to avoid additional packet delay due to backlog. These differences indicate that LAP should be used in an active network, where processing is performed on a best-effort basis and no explicit flow setup is performed. EFQ on the other hand requires explicit reservations, which are only available in a network that is tightly managed and where resources are controlled. Such a network could provide QoS in terms of bandwidth and processing resources.

To combine these scheduling principles on a system, the following points need to be considered. First, it is necessary to maintain data structures for both scheduling algorithms in the system to allow quick transitions between the schedulers. Second, a policy has to be defined under which circumstances these transitions happen. As for the data structures, it is necessary to maintain per-flow queues for EFQ to maintain QoS guarantees. At the same time the LAP data structure can track the individual packets in terms of application groups. This could cost in the worst case twice the amount of memory for control data structures, as a queue is necessary for each flow and each application. Also the enqueue and dequeue operations take twice as many memory accesses, because a packet has to be enqueued and dequeued in two data structures. Using two parallel memories might allow the parallelization of this operation.

The decision to switch from EFQ to LAP scheduling or back can be based on several factors. In a network with admission control, it should never happen that the NP is permanently overloaded. But due to the nature of general-purpose processing, the reservations for processing resources are based on heuristics (i.e., processing time estimation). It could happen that a NP becomes temporarily overloaded by a set of packets that require unusually complex processing. In such a case, it might be desirable to switch to LAP to increase the system performance. An event that could trigger such a switch could be when the overall amount of queued packet data exceeds a certain threshold. A more accurate trigger could be the exceeding of a threshold of overall estimated processing time. Similarly, if the overall amount of queue data goes below another threshold, the system could switch back to EFQ.

It should be noted that in the event of a switch to LAP, the original QoS guarantees (e.g., bounded delay) can not be met anymore. However, by sacrificing the guarantees temporarily, the system can get back to a lower level of load and therefore sooner return to QoS operation.

It might be possible to integrate the ideas of LAP directly into EFQ. In EFQ, we assume that the processing resource is not oversubscribed. Therefore, it is quite possible that at any point of time when a scheduling decision is necessary more than one processor is available to handle the scheduled packet. In such a case, locality information could be used to chose a processor, which has a warm instruction cache. This would reduce the processing time of a packet and allow more flows to be admitted to the router.

7. Related work

Most software-based programmable routers 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 [19] asks for packets to be processed by individual threads to allow for an accounting mechanism. However, methods for admission control and QoS scheduling are not described. Qie et al. [21] describe the problem of scheduling computational resources among competing flows, but relies on being able to pre-determine the processing time of packets. Also, the 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 [17], which limits its 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 [28]. The performance of these disciplines has been compared to GPS [18], 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^{2}Q [1] 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 RPS [24], 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 SFQ [14], which has been deemed suitable for CPU scheduling [13]. 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 [2]. 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.

Cache-affinity scheduling, which uses locality information for the scheduling decision has been used mostly in shared memory multiprocessors [10,23,25,26]. The focus in this domain is to schedule the same process or thread on processors that can reuse previously established cache states. While this is similar to the NP environment, it does not consider the reuse of instruction cache state for different threads that use the same instruction code (as is done with packets that use the same application).

An example for scheduling that uses hints about the processing requirement is [20]. In this work, the compiler provides information about thread requirements that are used by the scheduler to determine a thread execution schedule with high cache locality.

Salehi et al. show the effect of affinity-based scheduling on network processing in [22]. While this also considers the processing of network traffic, the focus is on the operating system level, where packet processing is disrupted by a background workload. This switching between packet processing and the background workload reduces locality in execution and can be avoided by appropriate scheduling.

8. Summary

This work proposes to use processing time estimations for packets in scheduling to achieve QoS guarantees and efficient operation of NPs. To address the problem of theoretically unpredictable processing times, we present measurements of processing times on a NP system, which indicate that processing on an NP is highly regular and predictable. Therefore it is possible to use

processing time predictions in admission control and scheduling decisions.

We present two such predictive processor scheduling algorithms that operate in two different operational regimes of networks. EFQ is designed for QoS networks and can provide guarantees on fair processor sharing and delay bounds. LAP is designed for best-effort networks and can reduce the number of cold cache events and increase the overall processing throughput. For EFQ an extensive analysis shows its superior behavior as compared to naive SFQ. For both algorithms, simulation results are presented that show the performance and performance tradeoffs.

The predictive processor schedulers presented in this work can provide fairness, delay bounds, and efficient operation, which is an important first step to integrating the flexibility of programmable routers into networks that require performance guarantees.

References

[1] J. Bennett, H. Zhang, Worst case fair weighted fair queuing, in: Proceedings of IEEE INFOCOM 95, Boston, MA, April 1995, pp. 120–128.

[2] J.C.R. Bennett, H. Zhang, Hierarchial packet fair queuing algorithms, in: Proceedings of ACM SIGCOMM, Palo Alto, CA, August 1996, 43–56.

[3] J.M. Blanquer, B. Ozden, Fair queuing for aggregated multiple links, in: Proceedings of ACM SIGCOMM 2001, San Diego, CA, August 2001.

[4] T. Chaney, A. Fingerhut, M. Flucke, J. Turner, Design of a gigabit ATM switch, in: Proceedings of IEEE INFOCOM 97, Kobe, Japan, April 1997.

[5] Y. Chen, J.S. Turner, Design of a weighted fair queueing cell scheduler for ATM networks, in: Proceedings of IEEE GLOBECOM 98, Sydney, Australia, November 1998.

[6] S.Y. Choi, J.S. Turner, T. Wolf, Configuring sessions in programmable networks, in: Proceedings of the Twentieth IEEE Conference on Computer Communications (INFOCOM), Anchorage, AK, April 2001, pp. 60–66.

[7] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner, Router Plugins—a modular and extensible software framework for modern high performance integrated services routers, in: Proceedings of ACM SIGCOMM 98, Vancouver, BC, September 1998.

[8] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, B. Plattner, A scalable, high performance active network node, IEEE Network 31 (1) (1999) 8–19.

[9] J.D. DeHart, W.D. Richard, E.W. Spitznagel, D.E. Taylor, The smart port card: An embedded UNIX processor architecure for network management and active networking, Technical report WUCS-01-18, Department of Computer Science, Washington University, St. Louis, August 2001.

[10] M. Devarakonda, A. Mukherjee, Issues in implementation of cache-affinity scheduling, in: Proceedings of Winter USENIX Conference, January 1992, pp. 345–357.

[11] V. Galtier, K.L. Mills, Y. Carlinet, S. Leigh, A. Rukhin, Expressing meaningful processing requirements among heterogeneous nodes in an active network, in: Proceedings of the Second International Workshop on Software and Performance, Ottawa, Canada, September 2000.

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

[13] P. Goyal, H.M. Vin, H. Cheng. A hierarchial cpu scheduler for multimedia operating systems, in: Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI), Seattle, WA, October 1996, pp. 107–121.

[14] P. Goyal, H.M. Vin, H. Cheng, Start-time fair queuing: a scheduling algorithm for integrated services packet switching networks, in: Proceedings of ACM SIGCOMM, ACM, Palo Alto, CA, August 1996, pp. 157–168.

[15] IBM Corp. IBM Power Network Processors, 2000. Available from <http://www.chips.ibm.com/products/wired/ communications/network_processors.html>.

[16] Intel Corp. Intel IXP2800 Network Processor, 2002. Available from <http://developer.intel.com/design/network/ products/npfamily/ixp2800.htm>.

[17] J.T. Moore, M. Hicks, S. Nettles, Practical programmable packets, in: Proceedings of IEEE INFOCOM 2001, Anchorage, AK, April 2001, pp. 49–59.

[18] A.K. Parekh, R.G. Gallager, A generalized processor sharing approach to flow control: the single node case, in: Proceedings of IEEE INFOCOM 92, Florence, Italy, May 1992, pp. 915–924.

[19] L. Peterson (Ed.), NodeOS interface specification, Technical report, AN Node OS Working Group, January 2001.

[20] J. Philbin, J. Edler, O.J. Anshus, C.C. Douglas, K. Li, Thread scheduling for cache locality, in: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, October 1996.

[21] X. Qie, A. Bavier, L. Peterson, S. Karlin, Scheduling computations on a software-based router, in: Proceedings of IEEE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Cambridge, MA, June 2001, pp. 13–24.

[22] J.D. Salehi, J.F. Kurose, D. Towsley, The effectiveness of affinity-based scheduling in multiprocessor networking, in: Proceedings of IEEE INFOCOM 96, San Francisco, CA, March 1996.

[23] M.S. Squillante, E.D. Lazowska, Using processor cache affinity information in shared-memory multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems 4 (2) (1993) 131–143.

[24] D. Stiliadis, A. Varma, Rate proportional servers: a design methodology for fair queuing algorithms, IEEE/ACM Transactions on Networking 6 (2) (1998) 164–174.

[25] J. Torrellas, A. Tucker, A. Gupta, Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors, Journal of Parallel and Distributed Computing 24 (1995) 139–151.

[26] R. Vaswani, J. Zahorjan, The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors, in: Proceedings of Thirteenth Symposium on Operating Systems Principles, Pacific Grove, CA, October 1991, pp. 26–40.

[27] T. Wolf, M.A. Franklin, CommBench—a telecommunications benchmark for network processors, in: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, April 2000, pp. 154–162.

[28] H. Zhang, Service disciplines for guaranteed performance service in packet switching networks, Proceedings of the IEEE 83 (10) (1995) 1374–1396.

Prashanth Pappu is a doctoral student in the Applied Research Laboratory (ARL) in the Department of Computer Science and Engineering at Washington University in St. Louis. His research deals with the design, analysis and implementation of high capacity, high performance routers. He received his Master's degree in Computer Science from Washington University in 2002 and his Bachelor of Technology degree in Computer Science from Indian Institute of Technology (Madras) in 1999.

Mark A. Franklin received his B.A., B.S.E.E. and M.S.E.E. from Columbia University, and his Ph.D. in EE from Carnegie-Mellon University. He is currently at Washington University in St. Louis where he is in the Department of Computer Science and Engineering, and is the Hugo F. and Ina Champ Urbauer Professor of Engineering. He founded the Computer and Communications Research Center and, until recently, was the Director of the Undergraduate Program in Computer Engineering.

Dr. Franklin is engaged in research, teaching and consulting in the areas of computer and communications architectures, ASIC and embedded processor design, parallel and distributed systems, and systems performance evaluation.

He is a Fellow of the IEEE, a member of the ACM, and has been an organizer and reviewer for numerous professional conferences including the Workshops on Network Processors 2002 and 2003. He has been Chair of the IEEE TCCA (Technical Committee on Computer Architecture), and Vice-Chairman of the ACM SIGARCH (Special Interest Group on Computer Architecture).

Tilman Wolf received his Diplom in Informatics from the Universität Stuttgart, Germany in 1998. From Washington University in St. Louis, he received a M.S. in Computer Science in 1998, a M.S. in Computer Engineering in 2000, and a D.Sc. in Computer Science in 2002. He is currently Assistant Professor in the Department of Electrical and Computer Engineering at the University of Massachusetts at Amherst. His research interests are advanced computer networks, programmable routers, network processor design, and benchmarking.