Controlling software router resource sharing by fair packet dropping

Vamsi Addanki, Leonardo Linguaglossa, James Roberts and Dario Rossi

Telecom ParisTech, Paris, France – `first.last@telecom-paristech.fr`

Abstract—The paper discusses resource sharing in a software router where both bandwidth and CPU may be bottlenecks. We propose a novel fair dropping algorithm to realize per-flow max-min fair sharing of these resources. The algorithm is compatible with features like batch I/O and batch processing that tend to make classical scheduling impractical. We describe an implementation using Vector Packet Processing, part of the Linux Foundation FD.io project. Preliminary experimental results prove the efficiency of the algorithm in controlling bandwidth and CPU sharing at high speed. Performance in dynamic traffic is evaluated using analysis and simulation, demonstrating that the proposed approach is both effective and scalable.

I. INTRODUCTION

Controlling how bandwidth is shared between concurrent flows is a classical issue in networking. While there are multiple objectives in this field and many proposed mechanisms, we concentrate in this paper on max-min fair sharing between a dynamically changing population of flows in progress.

The advantages of imposing bandwidth fairness have been repeatedly discussed since Nagle's pioneering observations [22]. See [23, Sec. 7] for a very clear summary. Satisfactory performance is maintained even when end-systems do not comply with TCP-like congestion control. More efficient high speed transport protocols can be introduced without requiring them to be friendly to legacy TCP. Implicit service differentiation is realized in that low rate streaming flows naturally experience negligible packet loss and delay.

In emerging high-speed software routers, flow throughput may additionally be impeded by resources other than bandwidth. We concentrate in this paper on the CPU executing virtualized network functions for packet forwarding and processing. CPU capacity is measured in cycle/s and flows may differ widely in their per-packet requirements depending on the functions they execute. The considered objective here is max-min fair flow rates in cycle/s, the product of the packet/s rate and the number of cycles needed to process each packet.

In Ghodsi *et al.* [12], fair bit/s bandwidth sharing and fair cycle/s CPU sharing are coupled in the notion of dominant resource fairness (DRF). In this work, we propose rather to control fair sharing of bandwidth and CPU resources independently (i.e., without using weights that depend on the dominant resource). This is both simpler to implement than DRF and fulfils a multi-resource sharing objective that is in significant ways preferable [8].

The mechanism envisaged in [12] and [8] for imposing fair shares is a scheduler like start time fair queuing (STFQ) [13]. However, this approach is hardly compatible with the hardware and software optimizations that are necessary to keep up with line speeds of 10 Gbps and more on a single CPU core. These optimizations notably require packets to be batched for both I/O and processing making implementation of classical scheduling algorithms problematic if not impossible, as argued in [28]. We therefore propose a more flexible software oriented solution based on fair packet dropping.

A number of approximate fair dropping algorithms have already been proposed for fair bandwidth sharing, such as FRED [17], CHOKe [24], RED-PD [21] and AFD [23]. In a preliminary evaluation, we found these algorithms imprecise and difficult to implement, especially in the present context of a software router. We have preferred to explore an original *exact* fair dropping algorithm. This algorithm is shown to be scalable since it operates only on the limited number of flows that would currently be backlogged in a fair queuing scheduler [18].

Despite strong current interest in network function virtualization, there is still little published work on how one might control CPU sharing between concurrent flows. A recent paper by Vasilescu *et al.* recognizes the need for fair sharing and advocates a differential congestion marking scheme to account for flows with different cycle/packet costs [29]. Shin *et al.* have previously advocated a similar congestion marking scheme [26]. Marking is less robust than fair dropping since, to achieve fairness, it is necessary that end-systems respond correctly to the congestion notification. In environments where this is a reasonable assumption, our proposed algorithm could be trivially modified to perform accurate fair marking instead of fair dropping.

Our main contributions here are to define an original algorithm to control CPU sharing using fair dropping and to evaluate its performance by analysis, simulation and experimentation. Application of the same approach to control bandwidth sharing is also novel but does not differ radically from earlier proposals. Successive presentation of algorithms and results for bandwidth sharing and CPU sharing usefully highlights the additional complexity in controlling the latter.

We first discuss salient features of software routers (Sec.II) before introducing the proposed fair dropping algorithms and illustrating their behavior by simulation (Sec. III). A prototype implementation in the FD.io software router is
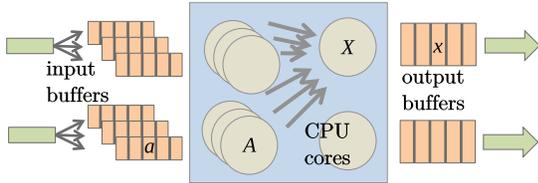
Fig. 1. Sketch of software router: core A forwards flows from input buffer a, core X handles all packets destined to output x.

then described and experimentally evaluated (Sec.IV). Finally, simulations in dynamic traffic confirm the scalability of the proposed approach and demonstrate its throughput performance (Sec. V).

II. SOFTWARE ROUTERS

We highlight features of emerging high-speed software routers that are significant for controlled resource sharing. For a more complete discussion, see [1], [5], [11], for instance.

A. Throughput bottlenecks

Flow throughput may be momentarily impeded by multiple bottlenecks in a software router. We consider here just two, output link bandwidth and CPU forwarding capacity. Note that CPU forwarding includes basic forwarding operations but also more complex tasks like encryption or a range of virtualized network functions.

Controlling bandwidth sharing by scheduling and buffer management is a classical function in networking with many proposed solutions. In software routers a notable example is the DPDK QoS framework that includes a variety of mechanisms ranging from the token bucket to a hierarchical weighted fair queuing scheduler [2]. These mechanisms are necessarily implemented in a CPU core that sees all packets destined for a given output port. In Fig. 1, the core in question for output port x is labelled X. The fair dropping algorithm we propose would similarly be implemented in core X (of course, the same core could process multiple outputs). It is an efficient alternative to scheduling.

The flexibility of software routers means packet processing capacity can be more closely matched to demand than in a hardware router and CPU can be a bottleneck. In Fig. 1 core A handles flows from input buffer a but, depending on demand, it might handle more buffers from multiple NICs. A CPU core becomes a bottleneck when flows emit packets too fast yielding a compute load greater than capacity and leading therefore to packet drops. In the absence of any control, the most aggressive flows will seize more CPU cycle/s than the others. Unfairness is exacerbated by the unequal per-packet costs of flows with different protocols [1] or middlebox function requirements [12]. We demonstrate that a lightweight fair dropping mechanism ensures flows get their fair share of CPU cycles. Implementing a scheduler like DRR [27] or STFQ [13] in this context, on the other hand, appears problematic to say the least.

B. Flow-awareness

High-speed software routers are intrinsically flow-aware. Flow-awareness is facilitated by NICs implementing receive side scaling (RSS). RSS performs a hash of packet header fields (e.g., the usual 5-tuple) and maps this to distinct queues, mainly for the purpose of load balancing over multiple CPU cores. Individual threads of packet processing applications are bound to a CPU core and, using kernel-bypass stacks (such as DPDK [3], netmap [25], PF_RING [10] or pfq [9]), threads consume independent streams of packets, each from a different RSS queue. In Fig. 1, the splitting of incoming traffic over multiple input buffers is flow-aware. Importantly for the implementation of flow-aware functionality, the hash is recorded in packet metadata and can be accessed by router software.

A significant advantage of flow-awareness in a software router is that all packets of the same flow are processed by the same core bringing efficiencies and enabling limited per-flow state. FIB lookup efficiency is enhanced, for instance, since only the first packet of a flow will typically require a RAM memory access while the result will remain in cache for subsequent packets. Flow state is necessary for mechanisms like the present fair dropping proposal.

C. Batch-mode processing

It is significant that high-speed software routers and their NICs generally deal with packets in *batches* rather than individually. This is a necessary optimization for line-speed I/O and greatly improves the efficiency of packet processing. In kernel bypass stacks, batching significantly reduces interrupt pressure compared to per-packet operation. The CPU handling an input buffer therefore typically polls for available packets. It visits the buffer, grabs all waiting packets up to a maximum number and processes the entire batch before returning to grab another batch.

The most recent high-speed software routers also perform forwarding tasks successively on batches of packets [16], [5], [1]. Each task in the processing graph is performed on all packets in the batch before the CPU moves to the next task. Batched processing optimizes the use of the CPU instruction cache as code for a task only needs to be fetched once for the entire batch. In addition, the overhead of managing graph traversal (counters, pointers, calls,...) is minimized since most operations need be performed once only for the entire batch. Processing efficiency improves with the size of the batch and mechanisms beyond simple polling are generally employed to ensure the batch is large enough. Batching makes it impossible to implement classical schedulers that rely on dequeue operations being triggered by individual packet departures [28].

III. FAIR DROPPING

We present the fair dropping algorithm and numerically illustrate how it realizes per-flow fair bandwidth and CPU sharing under static demand.

A. Fair rates by dropping

Suppose packets are handled simultaneously by two service systems, one the actual buffer management system implemented in the router (e.g., FIFO), the other a shadow system implementing a more sophisticated scheduler (e.g., per-flow FQ). Packets that are dropped in one system are also dropped by the other so that both systems yield exactly the same rate over the lifetime of a flow.

The shadow system in our proposal is virtual and makes dropping decisions based on a measure of per-flow virtual queue occupancy. This measure is depleted between packet arrivals, at a rate that varies depending on the number of active flows, and incremented by packet length on the arrival of every batch. If we correctly track the virtual queue occupancies at arrival instants, and make drop decisions in the shadow system aggressively enough to avoid additional drops due to buffer overflow in the real system, flow rates are entirely determined by the shadow system. In particular, if the shadow system implements per-flow head-of-line processor sharing, the long-term flow rates will be max-min fair.

Fair dropping, as a software solution, is inherently more flexible than scheduling. In particular, the shadow system can be readily turned off when not needed, economizing CPU usage. If the sum of rates of flows in progress is currently less than the capacity of the resource in question, there is no need to impose fairness. This may be the usual case (e.g., for a backbone link, or a CPU that only performs forwarding) with fair dropping ready to be turned on as necessary (e.g., when a high rate server-to-server flow starts up, or a new IPSec flow suddenly saturates the CPU).

B. The algorithms

Algorithm 1 is used for fairly sharing either bandwidth or CPU capacity. It uses a table called ActiveList, containing the current virtual queue size ($flow.vq$) for all backlogged flows (i.e., $flow.vq > 0$) indexed by identifiers $flow$ (typically a hash of header fields). Following the arrival of a batch of packets, ActiveList is updated using lines 3 to 15. Virtual queue occupancies $flow.vq$ are reduced by their max-min fair share of service capacity accumulated since the last call. If $flow.vq$ goes to zero, the flow is removed from ActiveList.

Lines 16 to 28 deal with the newly arrived packet or packets. Packets are dropped if the virtual queue of their flow exceeds a threshold θ. New flows are added to ActiveList and virtual queues are incremented by the size of the new packet. For bandwidth sharing $packet.length$ is measured in bytes while for CPU sharing it is an estimate of the number of cycles needed to process the packet.

For bandwidth sharing Algorithm 1 is sufficient and must be performed in a CPU receiving all packets destined to the considered output link. For CPU sharing, it is necessary to perform additional instructions to account for the fact that $packet.length$, the number of cycles needed to process the packet is not known *a priori*. Moreover, the number of cycles used to process a batch includes an overhead accounting for the cycles expended on dropped packets.

Algorithm 1 Virtual queue updates and dropping performed on arrival of a batch of packets.

1: **Given**: Δt - time since last update, C - service rate, $\mathcal{B}$ - ActiveList of backlogged flows, $\mathcal{P}$ - batch of new packets, θ - a threshold.
2: input $\mathcal{P}$
3: $credit = C\Delta t$
4: **while** $credit > 0$ **and** $|\mathcal{B}| > 0$ **do**
5:    $share = credit/|\mathcal{B}|$
6:    $credit = 0$
7:    **for** each $flow \in \mathcal{B}$ **do**
8:      **if** $share < flow.vq$ **then**
9:        $flow.vq \mathrel{-{=}} share$
10:      **else**
11:        $credit \mathrel{+{=}} share - flow.vq$
12:        $\mathcal{B} = \mathcal{B} \setminus flow$
13:      **end if**
14:    **end for**
15: **end while**
16: **for** each $packet \in \mathcal{P}$ **do**
17:    **if** $flow(packet) \in \mathcal{B}$ **then**
18:      **if** $flow.vq > \theta$ **then**
19:        drop $packet$
20:        $\mathcal{P} = \mathcal{P} \setminus packet$
21:      **else**
22:        $flow.vq \mathrel{+{=}} packet.length$
23:      **end if**
24:    **else**
25:      $\mathcal{B} = \mathcal{B} \cup flow$
26:      $flow.vq = packet.length$
27:    **end if**
28: **end for**

Algorithm 2 Cost calculation for CPU sharing for flows of different types.

1: **Given**: $\mathcal{P}$ - batch of packets, Δc - cycles consumed to process $\mathcal{P}$, w_k - relative weight of type k
2: $sumw = \sum_{packet \in \mathcal{P}} w_{type(packet)}$
3: **for** each $packet \in \mathcal{P}$ **do**
4:    $packet.cost = (w_{type(packet)}/sumw)\Delta c$
5:    $flow.vq = flow.vq + packet.cost - packet.length$
6: **end for**

Algorithm 2 apportions the measured overall batch processing cost Δc in proportion to weights giving the relative number of cycles needed to process packets of given type (e.g., a packet needing to consult an ACL may need more than ten times as many cycles as a packet that is simply forwarded). This cost calculation can only be performed after processing is complete while Algorithm 1 uses an assumed $packet.length$ to make drop decisions. The latter might be an average cost estimate derived by prior measurement or a real time updated average based on $packet.cost$ estimated by Algorithm 2 for packets of the given type for previous

		Utilization breakdown	Latency
FQ	TD	0.59 / 0.35 / 0.06	47.2 / 3.54 / 2.03
FIFO	TD	0.59 / 0.35 / 0.06	29.2 / 29.1 / 28.9
FQ	FD	0.45 / 0.45 / 0.10	22.1 / 19.6 / 2.70
FIFO	FD	0.45 / 0.45 / 0.10	18.9 / 18.9 / 19.4

Fig. 2. Dropper for CPU sharing is placed after the buffer but uses times packets arrive at the buffer.

batches. The virtual queue lengths must be corrected (line 5) to ensure they accurately track the actual numbers of expended cycles. Algorithm 2 can be performed at the start of a new processing cycle, before Algorithm 1, using data for the packets $\mathcal{P}$ processed in the previous cycle. Δc is the total number of cycles consumed between successive polling events.

Algorithm 1 realizes per-flow max-min fairness. It could easily be extended to realize more general objectives like hierarchical weighted fairness [7], for instance. Parameters identifying flow classes and weights would be stored in the flow table along with current virtual queue lengths. These would be used to derive flow specific shares in place of the common value computed in line 5.

C. Scalability

It is commonly believed that per-flow fair scheduling is not scalable since compute time grows with the number of flows and this number can attain many thousands on high speed links. This reasoning would apply similarly to Algorithms 1 and 2. In fact, the algorithms *are* scalable since, though the number of flows *in progress* may be very large, the set of *active* flows $\mathcal{B}$ includes only those that currently have a backlog. Under reasonable assumptions about the stochastic process of flow arrivals this number is small with high probability even for very high speed links, as demonstrated analytically and by trace driven simulation in [18].

The underlying analytical model is a processor sharing (PS) system with Poisson flow arrivals that has simple and robust performance characteristics [6]. Let ρ denote the PS server load, $\rho =$ flow arrival rate $\times$ flow size / server capacity, with $\rho < 1$ for stability. The number of active flows has a geometric distribution (i.e., $\mathbb{P}[\text{active flows} \geq x] = \rho^x$) and the expected completion time of a flow of size s is $s/(1-\rho)$. As a measure of flow throughput we use the reciprocal of the normalized flow completion time, $(1-\rho)$. These results apply for a wide range of stochastic demand models, as discussed in [6]. In particular, they do not depend on any assumption about the distribution of flow size.

Scalability in the present context is demonstrated by simulation in Sec. V while in this and the following section we illustrate algorithm performance for static sets of concurrent flows.

D. Bandwidth sharing

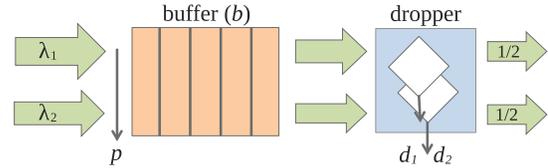For bandwidth sharing, fair dropping can be implemented before buffering packets at the output port. Bandwidth shar-

ing is then as fair as that realized with fair queuing schedulers like DRR [27] or STFQ [13].

Table I illustrates through simulation results for a toy example, that FQ scheduling is *not necessary* for fairness while FD is *essential*. Three flows emit unit size packets as a Poisson process at respective rates 1, 0.6 and 0.1 and share a unit rate link using FQ or FIFO. Excess packets suffer tail drop (TD) in the buffer preceding the link, or fair drop (FD) using Algorithm 1. ActiveList updates are performed here after every packet arrival. Buffer capacity is 30 packets and, for fair dropping, we set threshold θ to 10. Latency is measured in packet transmission times.

The results confirm that FD is sufficient for fair throughput while significantly increasing the latency of the low rate third flow. The negative impact of higher latency, for VoIP applications say, can be removed by replacing the link FIFO by a priority scheduler where packets belonging to flows absent from ActiveList on their arrival are served first [19], [14]. Latency could also be reduced by operating the drop algorithm with a rate C somewhat less than the link rate (e.g., as in [4]).

E. CPU sharing

To share CPU capacity, dropping can only take place after the input buffer. To emulate a fair scheduler, the shadow system must however make drop decisions based on packet arrival times at the buffer as recorded in a time stamp. These 'fair drops', based on the size of the per-flow virtual queues, are in addition to any 'tail drops' due to buffer saturation. The relative proportions of tail drops and fair drops depends on the choice of threshold θ. It is also necessary here to account for the fact that the act of dropping a packet consumes CPU cycles that are otherwise to be shared fairly.

a) An adaptive threshold: To illustrate how fair dropping realizes fair CPU sharing, consider the toy example of Fig. 2. N flows ($N = 2$ in the figure) bring processing requirements λ_i (packet/s $\times$ cycle/packet) and are served by a unit capacity CPU. A fraction p of each flow is lost due to buffer overflow. The remaining $(1-p)$ fractions of each flow share the CPU in max-min fashion, thanks to fair dropping, and suffer additional drops at rates d_i.

This queuing system is particularly complicated and we have no analytic results to determine the impact on p and the d_i of particular choices of buffer size b and drop threshold θ. Note, however, that while b is system dependent and fixed, θ is simply a program parameter and can be set and reset as
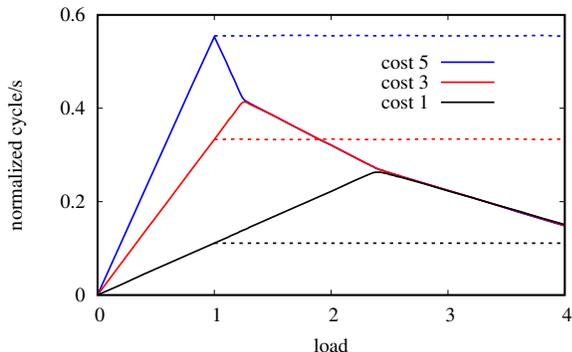
Fig. 3. CPU flow throughput as a function of load for 3 flows with equal packet/s rates and respective CPU costs 5, 3 and 1 and drop overhead 0.5; dotted lines show throughput without fair dropping.

necessary. It is possible, in particular, to adapt θ from one batch to the next based on observations of a performance objective.

A first performance objective is to make the probability of buffer saturation negligibly small. This is necessary to avoid undue loss for flows emitting at a rate less than the fair rate or flows consisting of a single packet like DNS queries. A second objective is to maintain processing efficiency by making batches as large as possible (cf. Sec. II-C). We therefore adapt θ as follows: if the polled vector is maximal (suggesting impending saturation), multiply θ by a factor $\alpha < 1$ to induce more fair drops; if the polled vector is not maximal, multiply θ by $\beta > 1$. The choice of parameters α and β is not highly critical. In our simulations, setting $\alpha = .5$ and $\beta = 1.2$ gave satisfactory results.

b) Dropping overhead: Any dropped packet consumes cycles and this overhead reduces flow throughput. The overhead includes the cycles used to run Algorithms 1 and 2 but is mainly due to the cost of bringing packets into the CPU, as is necessary to determine their flow identity.

Fig. 3 presents per-flow cycle/s throughput for a toy system with 3 flows, each emitting packets as a Poisson process at the same rate, as a function of load. The flows have different relative per-packet processing requirements: flow 1 packets cost 5 units, flow 2 packets 3 and flow 3 packets 1 (the value of the cost unit in compute cycles is not significant, only their ratio matters). Dropped packets consume 0.5 units of CPU. Offered load on the x axis is the sum of the products (packet rate $\times$ cost) divided by the CPU capacity (i.e., $\sum \lambda_i$).

The figure plots normalized cycle/s throughputs (the sum of throughputs is 1 at load 1) against load. Dotted lines show throughputs realized in the absence of fair dropping. These results confirm that fair dropping realizes max-min fairness, e.g., the cost 1 flow suffers no loss until its input rate exceeds the fair rate when all flows have the same throughput. On the other hand, the sum of throughputs decreases with increasing load due to the cost of dropping. Throughputs go to zero at load 6 when the CPU is entirely busy dropping packets. Throughput can be bounded away from zero by setting a

minimum threshold at the cost, however, of significant loss for low rate and single-packet flows.

IV. IMPLEMENTATION

We have implemented the fair dropping algorithms on a real software router. In the following we describe the testbed, outline the software architecture and present experimental results.

A. Experimental setup

Algorithms 1 and 2 run in the Vector Packet Processing (VPP) software router that is part of the Linux Foundation's FD.io project [1]. The VPP router is deployed in a server platform based on two Intel Xeon E52690 processors, each with 24 cores running at 2.60 GHz, equipped with two 10 Gbps Intel X520 NICs directly connected with SFP+ interfaces. The two processors and two line cards are isolated to logically create two independent nodes. This is realized using non-uniform memory access, to make portions of RAM accessible to only one line card, and CPU core binding where particular cores are mapped to a specific process and interrupts are deactivated.

Of the two nodes, one acts as traffic generator and sink (TGS) while the other is the system under test (SUT), the software router equipped with our FD algorithms. The TGS continuously sends a stream of packets to the SUT which processes them and sends them back to the TGS. We can measure the throughput of the SUT as the return input rate to the TGS. To validate the FD algorithm and its scalability, the TGS sends traffic consisting of 64-byte packets at 10 Gbps (corresponding to an input rate of 14.88 Mpps). IP addresses and port numbers are set to emulate a number of distinct constant rate flows. To stress the system, the FD algorithms are executed on a single core handling all traffic.

B. Software architecture

Our implementation[1] works on bandwidth and CPU sharing. The fair dropper is currently implemented within an FD.io node [1], but could alternatively be implemented as a lower-level primitive of the DPDK QoS framework [2]. Porting Algorithm 1 to DPDK is straightforward and would be sufficient for bandwidth sharing. However, per-packet cost estimates derived in Algorithm 2, as necessary for CPU sharing, must be made available to DPDK and this requires further work. Thus, while the experimental results presented below are specific to the FD.io implementation, they may be considered as a more general proof-of-concept for a software line-rate implementation of fair dropping.

Vector Packet Processing (VPP) [20], is a kernel-bypass application that *reads* and *processes* packets in batches. VPP consists of a set of software functions that logically abstract network operations of different layers of the protocol stack (examples of functions are `l2_input` or `ip4_lookup`). VPP links such functions to form a *processing graph* and

---

[1] https://github.com/TeamRossi/vpp-bench

each function (represented by a node in the graph) is applied to packets as necessary in order to implement packet processing and forwarding. Batched reads are performed by DPDK drivers accessing NIC hardware and significantly reduce interrupt pressure (I/O batching). Processing is also performed in batches of packets – called *vectors* – optimizing usage of the underlying CPU architecture. In other words, during graph traversal, each node function is applied to a full vector of packets (compute batching) before continuing to the next node. For bandwidth sharing, Algorithm 1 runs in the final output nodes of the VPP processing graph while, for CPU sharing, both algorithms are implemented in the initial node called `dpdk-input`.

In our implementation, FD operations are performed once per batch thus matching the typical work-flow of a VPP router. This reduces the induced overhead without unduly impacting realized fairness (cf. Sec. III). To reduce computational complexity, we identify the flow using the 5-tuple hash computed by the NIC for RSS queues. This is accessible via the `hash.rss` variable within the `mbuf` DPDK structure.

Flows are stored in a flow table. The data structure used is a hash-table with 4K rows each receiving up to 4 24-byte flow records. The row is addressed by 12 bits of the RSS hash and the flow record uses 8 more bits of the hash to distinguish up to 4 flows mapped to the same row. The row size is aligned to fit two cache lines in our platform. In view of the scalability results discussed in Sec. III-C, the flow table size is largely sufficient to ensure the probability of misidentifying a flow is negligible.

We additionally maintain a separate data structure identifying the *active flows*, that is, the flows that currently have a positive virtual queue. This structure has two roles: it identifies the small set of flows to which the FD algorithm must be applied, and it enables data for this set to be maintained in CPU L1 or L2 cache. Flow state in the present implementation is confined to the flow identifier and the current virtual queue length. However, space remains for additional state needed by more complex objectives, like hierarchical weighted fairness for instance.

Implementation of Algorithm 1 for CPU sharing requires access to packet arrival times using a timestamp. NIC time stamping is currently available through the DPDK `rxtx_callback` function. When the callback is executed, the Time Stamp Counter (TSC) is accessed[2] and the TSC register value is written to the DPDK `mbuf` user data field `udata64`.

C. Bandwidth sharing

To illustrate FD induced bandwidth sharing we generate a workload consisting of 20 flows with progressively decreasing arrival rates: the flow with rank 1 has an arrival rate 10 times higher than that of flow 20. We produce a bottleneck by rate limiting the output port to a fraction α of the input rate.
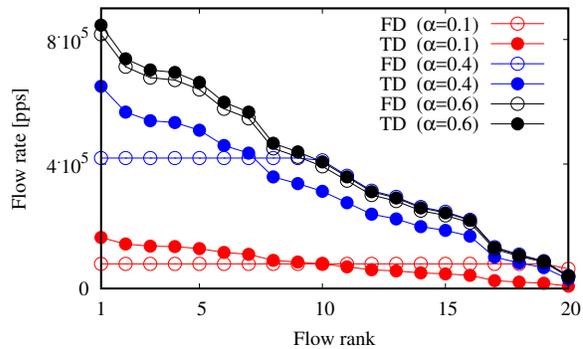

Fig. 4. Performance of bandwidth sharing: experiments for different output rates $\alpha \times 10$ Gbps (10Gbps line card, 1 CPU core, IPv4 forwarding, skewed packet rates).

Fig. 4 presents experimental results for 3 values of α. Per-flow output rates are plotted with either FD or TD (i.e., without differential dropping). With $\alpha = 0.1$, all flows are able to attain the fair rate under FD while rates are proportional to input rates under TD. With $\alpha = 0.4$ the available bandwidth increases and FD affects only those flows (of rank 1 to 10) that exceed the fair rate. In both cases, the overall drop rate under TD and FD is exactly the same, only *which* packets are dropped differs.

With $\alpha = 0.6$ the output link is no longer a bottleneck and flow rates are limited by the CPU processing capacity[3]. This implies flow rates are reduced proportionally due to input buffer saturation. The difference between the top two black lines in the figure is due to the overhead of our current non-optimized implementation of the FD algorithm (which doesn't actually drop any packets in this case).

D. CPU sharing

In our experiments on CPU sharing, the TGS creates 20 equal packet rate flows belonging to one of two different types: packets of type-L flows require "light" processing, while packets of type-H flows have a "heavy" cost, consuming r times more cycles than type-L. Per-packet cost depends on the amount of processing in the VPP graph for both I/O and computation and can be readily measured using VPP primitives [20]. In the experiments, 18 type-L flows send IPv4 packets requiring standard processing: longest prefix matching and next hop forwarding. Two high-cost flows additionally pass via a busy loop whose length can be precisely controlled to modulate the ratio r of H to L type costs.

Experimental results are represented as Sankey diagrams in Figure 5 where TD and FD are compared for $r = 10$ with an input rate of 14.88 Mpps. With tail drop, 11.02 Mpps are dropped at the NIC interface, and the rest is processed by the SUT. Tail drop makes no explicit decision as to which

---

[2]TSC is a 64-bit register whose purpose is to count the CPU clock cycles

[3]This corresponds to about 8.5 Mpps IPv4 forwarding throughput

(a) TailDrop

(b) FairDrop

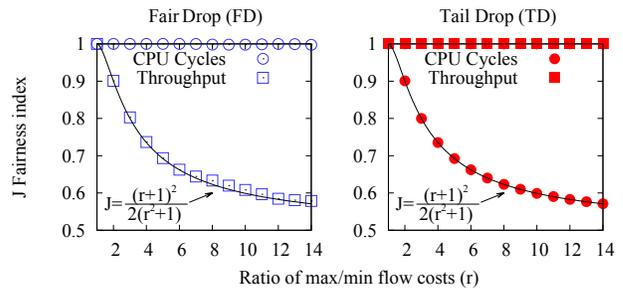Fig. 5. Sankey diagrams for Tail Drop (a) and Fair Drop (b) experiments

Fig. 6. Throughput and CPU fairness indices for FD (left) and TD (right) as a function of the ratio r of type-H to type-L costs.

average flow cycle/s and packet/s throughputs [15]. With 2 types and respective metrics x_L and x_H, the Jain index is $(x_L + x_H)^2/(2(x_L^2 + x_H^2))$. Results for FD and TD, with x representing cycle/s and packet/s throughputs, are shown in Figure 6. The figure confirms that FD fairly shares CPU cycles while TD is fair in terms of packets/s (only because the input rates are equal). In contrast, FD packet/s throughputs and TD cycle/s throughputs are unfair in the ratio r with index $(r + 1)^2/2(r^2 + 1)$.

V. PERFORMANCE IN DYNAMIC TRAFFIC

We evaluate throughput performance in dynamic traffic and demonstrate scalability by simulation.

A. Demand model

Traffic demand is modeled as a Poisson process of flows of finite size of different types. Packets have constant size in bytes. Per-packet processing cost depends on the flow type and is assumed constant for all packets of the same flow.

We distinguish full-rate flows and single-packet flows. Full-rate flows last until 30000 unit size packets are successfully transmitted. As noted in Sec. III-C, we expect performance to be insensitive to the size distribution. Constant size is chosen for faster convergence of the simulations. The stream of single-packet flows is intended to include traffic from flows emitting packets at a rate less than the typical fair rate, possibly because of other bottlenecks on their path. The packets of such flows appear to buffer management as a succession of distinct single-packet flows.

Rather than simulating a transport protocol like TCP, we suppose full-rate flows emit packets as a Poisson process. They continue emitting packets until the number of successfully transmitted packets equals the flow size. The Poisson rate may be large and constant when flows are assumed unresponsive to congestion. To represent responsive flows we assume the Poisson rate is set to a value somewhat larger (10% here) than the rate that would be realized by a hypothetical ideal transport protocol. This simplification facilitates the evaluation of salient features of the considered algorithms, as presented below.
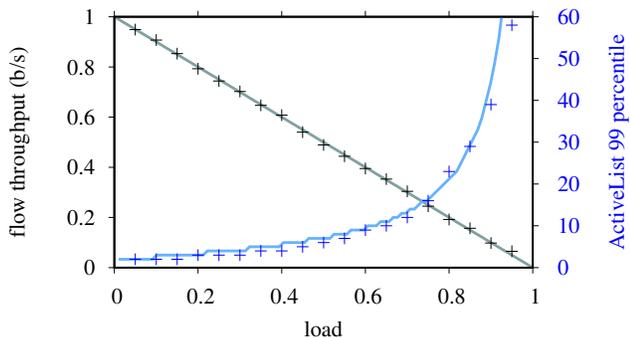
packets should be dropped and, since packet rates are equal among flows, drops affect equally the H and L classes (74%). It follows that all flows have the same bit rate while each of the H flows individually consumes 25% of the CPU cycles.

FD radically changes the operational point. The NIC drops 8.19 Mpps (55%) increasing the traffic processed by the SUT to 6.69 Mpps. The FD decision consumes 0.17 G cycle/sec (the overhead of the FD algorithm) and affects 0.74 Mpps of packets. As expected, 98% of the dropped packets are of type-H. This differentiation realizes fairness in terms of CPU cycles, since each of the 20 flows now receives exactly the 5% fair share of CPU. Notice also that, in this particular scenario, the overall rate of packets forwarded by the SUT increases: throughput is 5.95 Mpps with FD, compared to 3.86 Mpps with TD).

The drop threshold θ is fixed in these experiments. The packet arrival rate is such that it is not possible to eliminate buffer saturation. With respect to the overhead due to dropping packets, we measured the following average costs. To successfully send a type-L packet requires 350 cycles while a dropped packet costs 208 cycles made up of 120 for I/O, 50 for freeing packet memory and 38 for executing the drop algorithms.

To further illustrate the difference in fairness between FD and TD we repeat the above experiment with the value of r ranging from 1 to 14. For each value we compute the Jain fairness index between types L and H for both

Fig. 7. Performance of bandwidth sharing: normalized throughput and ActiveList 99th percentile with per packet updates; lines plot analytical results $(1 - \rho)$ and $\lceil -2/\log_{10} \rho \rceil$.
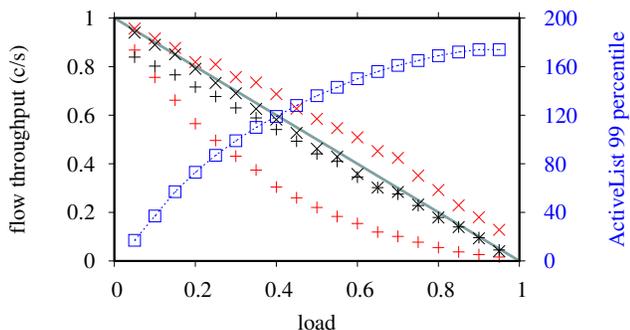

Fig. 8. Performance of CPU sharing: normalized cycle/s throughput with fair dropping (black) and tail dropping (red) for 2 flow classes, class 2 ($\times$) requires 10 times more cycles/packet than class 1($+$), relative cost of dropped packets 0.5; ActiveList 99th percentile for fair dropping (blue $\square$).

B. Bandwidth bottleneck

Let λ_f be the arrival rate of full-rate flows and λ_s the arrival rate of single-packet flows. The load of the unit capacity link is then $\rho = 30000\lambda_f + \lambda_s$. Given ρ, λ_f and λ_s are set so that full-rate flows contribute 80% of link load while single-packet flows make up the remaining 20%. Full-rate flows are unresponsive and emit packets at a constant rate such that total instantaneous demand largely exceeds link capacity.

Fig. 7 plots throughput (left y-axis) and the 99th percentile of the distribution of the number of flows in ActiveList (right y-axis) as functions of load ρ (flow arrival rate $\times$ mean flow size / link capacity). The measure of throughput is the ratio of average flow size to average flow duration (equal to $1 - \rho$ for processor sharing, as discussed in Sec. III-C). The 99th percentile of the PS model is $\lceil -2/\log_{10} \rho \rceil$. The close agreement between analysis and simulation confirms that fair dropping is an effective control for bandwidth sharing even when flows are unresponsive.

C. CPU bottleneck

To evaluate the effectiveness of fair dropping in sharing a CPU bottleneck we simulate a mix of single-packet flows with unit per-packet cost and two types of full-rate flows

with respective per-packet costs 1 and 10. The relative cost of dropping a packet of any type is 0.5. Single-packet flows contribute 20% of load while the full-rate flow types each contribute 40%. The buffer size is 512 packets and maximum batch size is 256. Fair dropping threshold θ is adaptive between 20 and 50000 cost units using multipliers $\alpha = .5$ and $\beta = 1.2$ (cf. Sec. III-E).

If the cost of dropping were null, our simulation results (not shown here) confirm that, as for bandwidth sharing, fair dropping yields a common cycle/s flow throughput equal to $1 - \rho$, even when all flows are unresponsive. Unfortunately, this is not the case when the drop overhead is not negligible. In dynamic traffic, at some point the number of active flows will attain a level at which the CPU is saturated even when all packets are dropped (cf. Sec. III-E). Flow throughput then goes to zero and cannot recover since flows in progress do not complete while new flows continue to arrive. The impact can be mitigated by imposing a minimum threshold θ but at the cost of significant tail drops affecting single-packet flows.

It is important to note that this instability would occur with any active queue management or scheduling algorithm that selectively drops packets within the CPU. To effectively control unresponsive flows it would be necessary to selectively discard packets *before* they are polled by the CPU. We intend in future work to investigate the possibility of piloting such a mechanism using the fair dropping algorithm to identify the unresponsive flows in question.

Fair dropping remains an effective means for controlling CPU sharing between *responsive* flows with different per-packet costs. When fair dropping is employed, we know concurrent flows are allocated the same cycle/s throughput. We therefore assume responsive flows emit packets at a rate such that their cycle/s rate (packet/s $\times$ cost/packet) is 10% greater than the current fair rate. When fair dropping is absent, all packet loss is through tail drop and concurrent flows experience the same drop rate. As flow packet rate is determined by this drop rate (e.g., by TCP congestion control), we therefore derive a common packet/s rate for flows such that the sum of cycle/s rates is 10% greater than capacity. The 10% excess is meant to approximately capture the impact of a transport protocol like TCP that progressively increases flow rate until drops occur.

Fig. 8 plots throughput on the left y-axis, for tail dropping (red) and fair dropping (black), and the ActiveList 99th percentile on the right y-axis for fair dropping as functions of load (ρ = flow arrival rate $\times$ mean flow cycles cost / CPU capacity). Throughput behavior is broadly as expected: fair dropping yields almost ideal PS throughput for both flow types while tail dropping severely degrades the performance of the flows with lower CPU cycle cost, especially at high loads.

To explain observed unfairness of FD at low load, consider the throughput of an isolated full-rate flow emitting packets at 10% above the nominal CPU rate. The drop rate d_1 for cost-1 flows would be such that $1.1(.5d_1 + (1 - d_1)) = 1$,

i.e., the drop rate is such that packet arrival rate $\times$ average cost is equal to CPU capacity. This yields $d_1 = 2/11$ and a corresponding flow throughput of 0.9. A similar calculation for cost-10 flows yields a throughput of 0.99. The loss rate for single-packet flows is negligible with fair dropping but rises to around 10% with tail dropping.

Results for the ActiveList 99th percentile are quite different to those of Fig. 7. This is due to batch processing and the impact of single-packet flows. All single-packet flows in a batch bring a new ActiveList flow (that will be removed on the next batch arrival). The number of such flows depends on the batch size and is added to the small number of active full-rate flows that is accurately predicted by the PS model. Fair dropping remains scalable in that the number of active flows remains very small compared to the possibly large number of flows in progress.

VI. Conclusion

Applying proposed fair dropping algorithms in a software router has been shown to realize per-flow fair sharing of both bandwidth and CPU. The algorithms are scalable because the number of flows to be managed is small (less than 200 with high probability at normal loads) whatever the link speed or CPU capacity. It is compatible with batch I/O and batch processing, optimizations which significantly impede the implementation of classical schedulers.

The algorithms have been successfully implemented in the VPP software router that is part of the FD.io Linux Foundation project. Preliminary experimental results show them to be effective and relatively lightweight in terms of induced overhead.

There is, however, a significant overhead involved in selectively dropping packets within the CPU, due mainly to packet I/O, whatever the algorithm employed. This overhead can compromise performance when flows are non-responsive to congestion. We are currently investigating extensions to our approach where non-responsive flows can be effectively dealt with before their packets are polled by the CPU.

References

[1] https://fd.io/wp-content/uploads/sites/34/2017/07/ FDioVPPwhitepaperJuly2017.pdf.

[2] http://dpdk.org/doc/guides/prog_guide/qos_framework.html.

[3] Data plane development kit. http://dpdk.org.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, 2012.

[5] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, 2015.

[6] S. Benfredj, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: A study of congestion at flow level. In *ACM SIGCOMM*, 2001.

[7] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM*, 1996.

[8] T. Bonald and J. Roberts. Multi-resource fairness: Objectives, algorithms and performance. In *ACM SIGMETRICS*, 2015.

[9] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *Passive and Active Measurement*, pages 64–73. Springer, 2012.

[10] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *ACM IMC*, 2010.

[11] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet io. In *ACM/IEEE ANCS*, 2015.

[12] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *ACM SIGCOMM*, 2012.

[13] P. Goyal, H. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Netw.*, 5(5):690–704, Oct 1997.

[14] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290, Jan. 2018.

[15] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. DEC Research Report TR-301, 1984.

[16] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *ACM European Conference on Computer Systems (EuroSys)*, 2015.

[17] W.-J. Kim and B. G. Lee. Fred-fair random early detection algorithm for tcp over atm networks. *Electronics Letters*, 34(2):152–154, Jan 1998.

[18] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In *ACM SIGMETRICS*, 2005.

[19] A. Kortebi, S. Oueslati, and J. Roberts. Implicit service differentiation using deficit round robin. In *Proceedings of ITC19*, 2005.

[20] L. Linguaglossa, D. Rossi, D. Barach, D. Marjon, and P. Pfiester. High-speed software data plane via vectorized packet processing. Tech report, 2017.

[21] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, pages 192–201, Nov 2001.

[22] J. Nagle. On packet switches with infinite storage. RFC 970, 1985.

[23] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM Comput. Commun. Rev.*, 33(2):23–39, Apr. 2003.

[24] R. Pan, B. Prabhakar, and K. Psounis. Choke - a stateless active queue management scheme for approximating fair bandwidth allocation. In *INFOCOM*, 2000.

[25] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[26] M. Shin, S. Chong, and I. Rhee. Dual-resource tcp/aqm for processing-constrained networks. *IEEE/ACM Trans. Netw.*, 16(2):435–449, Apr. 2008.

[27] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Comput. Commun. Rev.*, 25(4):231–242, Oct. 1995.

[28] K. To, D. Firestone, G. Varghese, and J. Padhye. Measurement based fair queuing for allocating bandwidth to virtual machines. In *ACM HotMiddlebox*, 2016.

[29] L. Vasilescu, V. Olteanu, and C. Raiciu. Sharing cpus via endpoint congestion control. In *Proceedings of the Workshop on Kernel-Bypass Networks*, KBNets '17, pages 31–36, New York, NY, USA, 2017. ACM.