RZ 3848 (# ZUR1307-022) 07/16/2013
Computer Science 10 pages

Research Report

Tandem Queue Weighted Fair Smooth Scheduling

Nikolaos Chrysos*, Fredy Neeser*, Mitch Gusat*, Rolf Clauberg*, Cyriel Minkenberg*, Claude Basso‡, and Kenneth Valk‡

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

‡IBM Systems & Technology Group,
Rochester, USA

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Africa • Almaden • Austin • Australia • Brazil • China • Haifa • India • Ireland • Tokyo • Watson • Zurich

Tandem Queue Weighted Fair Smooth Scheduling

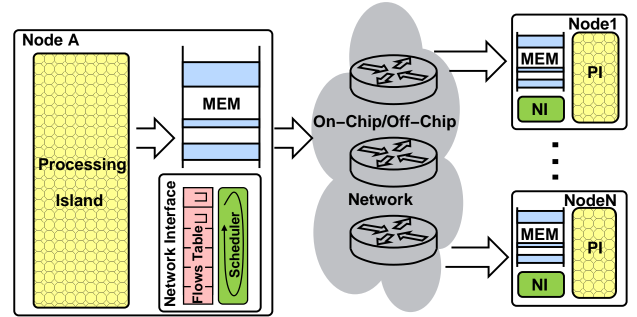
Nikolaos Chrysos, Fredy Neeser, Mitch Gusat, Rolf Clauberg, Cyriel Minkenberg
IBM Research – Zurich, Switzerland: {cry,nfd}@zurich.ibm.com
Claude Basso, and Kenneth Valk
IBM Systems & Technology Group, Rochester, USA: kmvalk@us.ibm.com

***Abstract*—Network devices supporting 100G links are in demand to meet the communication requirements of computing nodes in datacenters and warehouse computers. In this paper, we propose *TQ* and *TQ-Smooth*, two light-weight, fair schedulers that accommodate an arbitrarily large number of requestors and are suitable for ultra-high-speed links. We show that our first algorithm, TQ, as well its predecessor, DRR, may result in bursty service even in the common case where flow weights are approximately equal, and we identify applications where this can damage performance. Our second algorithm, TQ-Smooth, improves short-term fairness to deliver very smooth service when flow weights are approximately equal, while allocating bandwidth in a weighted fair manner. In many practical situations, a scheduler is asked to allocate resources in fixed-size chunks (e.g. buffer units), whose size may exceed that of (small) network packets. In such cases, byte-level fairness will typically be compromised when small-packet flows compete with large-packet ones. We describe a novel scheme that dynamically adjusts the service rates of request/grant buffer reservation to achieve byte-level fairness based on received packet sizes.**

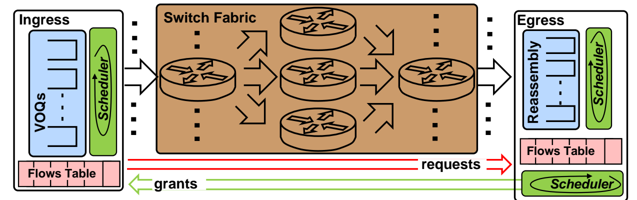
I. Introduction

Current datacenters (DCs) enclose many thousands of digital appliances capable of processing and storing massive amounts of data. When seen in isolation, these appliances are not always superior to what users may have at home. But the confinement of many of them within a small spot and the big-data applications that they can collectively engage in make datacenters particularly interesting. Datacenter networks (DCNs) are playing a critical compounding function in datacenters. In a still somewhat turbulent field, there have been many recent proposals to reshape DCNs so that the latter can successfully meet stringent and in some cases even divergent requirements. Many of these proposals focus on management, transport, or network level protocols, targeting better exploitation of the existing infrastructure by DC applications [2], [3], [4], [5], [6].

This is an extended version of a HiPEAC INA-OCMC paper [1], entitled "Arbitration of many thousand flows at 100G and beyond" by ACM, NY, USA ©2013, ISBN: 978-1-4503-1784-9, *doi>10.1145/2482759.2482761*



(a) Scheduling in a network interface, based on urgency, connection's window, tenant's subscription, etc.



(b) In a switching fabric with virtual output queues (VOQs), a packet scheduler may be needed *(i)* at ingress to prioritize packet injections, and at egress, *(ii)* to forward reassembled packets or *(iii)* to allocate egress buffer space for packets that wait at VOQs at ingress interfaces [9], [10].

Fig. 1. Possible applications of fast packet schedulers.

At the same time, the DCN infrastructure changes in ways that can radically modify the landscape. Intelligent network interfaces (NIs) attached to (or coexisting with) processing cores, which can provide low-latency / high-bandwidth pathways to remote processes, is a long sought goal –see Fig. 1(a) for an illustration, and refer to [8] for an example study.

Another trend is large, flattened switching fabrics (Fig. 1(b)) to deal with the increasing volume of inter-server (east-west) traffic. Such fabrics are expected to provide equidistant ports with homogeneous quality-of-service guarantees to enable the seamless integration of large numbers of compute and storage nodes.

Switches and NIs with 40G Ethernet ports are now becoming available, and the industry is already preparing

for 100G Ethernet. If any lesson has been learned from the past it is that bandwidth is rarely in excess. Although today there probably are only few processes that saturate a 100G link, this may not be the case in the near future. The adoption of RDMA and FCoE (Fibre Channel over Ethernet) technologies in converged datacenter networks will soon enable a single node to generate flows that can saturate even a 100G link. In addition, in a multi-VM, multi-tenant DCN environment, any link can easily become congested. Thus, it is a basic requirement for modern converged network architectures [7] to slice the capacity into well isolated traffic classes with performance guarantees.

On the other hand, scheduling becomes extremely challenging as network size and line speed increase. For instance, the duration of a 64B Ethernet frame on a 100G line is just ∼6 ns. At the same time, in a warehouse-scale computer, or in a multi-core processor, a scheduler may arbitrate among several hundreds to thousands of requestors. The trend towards large distributed switches and ultra-short packet durations make the implementation of a fair scheduler extremely difficult.

In this paper we propose a packet scheduling scheme that scales to large numbers of requestors and ultra-fast line rates. Packet schedulers have been studied extensively since the beginning of the ATM [11], [12], [13]. The simplicity vs. efficiency trade-offs that the *deficit round-robin (DRR)* introduced in the mid-1990s have rendered it the algorithm of choice in many operating systems and network devices [15].

Alternative schemes have since been proposed that maintain the same $O(1)$ asymptotic complexity of DRR while providing smoother service [16], [17], [18]. However, these algorithms add considerable bookkeeping and logic complexity. For modern ASICs, which may accommodate multiple high-speed links and support scheduling multiple flows in parallel, fair schedulers with low complexity are hence of great interest.

Our algorithm is built upon ideas from DRR:

a. it preserves the simplicity of the original DRR algorithm,
b. smoothes out burstiness, especially when the weights of the active flows are close to each other[1]
c. provably provides weight-proportional fair service.

For comparison, stratified-RR groups flows of roughly equal bandwidth requirements into a flow class [17]. It uses a timestamp-based scheduler to select a class, and DRR to select a flow within the class selected. While it is true that the timestamp-based scheduler arbitrates among significantly fewer candidates than the number of flows, its cost, coming on top of DRR, may be excessively high for ultra-high speed links. In addition, our DRR-based scheme easily integrates further improvements (Sec. IV) which would not be readily possible with a timestamp-based scheduler.

In Sec. II we describe our scheduling schemes and outline their behavior. Then, in Sec. III, we evaluate them using computer simulations. Section IV focuses on a particular application of the scheduler, namely, the allocation of credits of an egress buffer to a set of requestors in a large switching fabric. For this application, we describe how to maintain close to ideal byte-level fairness and point out some interesting trade-offs regarding the sizing of buffer subunits. Finally, Sec. V concludes our findings.

II. Ultra-Fast Packet Scheduling

In packet-switched networks, distributed packet schedulers are commonly responsible for slicing up the bandwidth. Weighted round-robin is a flexible scheduling scheme that extends the round-robin (RR) service by prioritizing requests using service weights. One can preconfigure or dynamically change weights to requestors, so that each weight assignment represents either the minimum bandwidth share (e.g. weights sum up to a 100) of each flow or the relative urgency/importance of one request over the other. We assume that flows are simply groups of requests. Each flow may present its demand either as a queue of unprocessed data packets or as a request counter of unprocessed data units. We disregard the possibility of two or more flows being merged into a common queue or counter.

We next present our two algorithms, *Tandem Queue (TQ)* and *Tandem Queue Smooth (TQ-Smooth)*. Their names originate from their control queues that work in tandem and propel their operation.

A. Tandem Queue: a DRR derivative

The scheduler keeps a configurable weight parameter $w_f \in N^+$ and a *service-credit* cr_f counter for each flow in memory. A descriptor of any eligible flow is always present in one of two *control queues*, which we name *highQ* and *lowQ*. Refer to Algo. 1. At any time, the scheduler selects the flow f at the head of one of those control queues, giving strict priority to highQ whenever it is non-empty. Note that the selected flow f is always removed from the corresponding control queue. Assume that the selected flow f is assigned L units of service, where L may correspond to the bytes of a head-of-line (HOL) packet or to a number of buffer units. Next, the scheduler *(a)* decrements cr_f by L, and *(b)* increments it

[1] In the common case, flow weights are likely to be equal.

by w_f if f was dequeued from lowQ. If f is still eligible after receiving this service or if it becomes eligible after an inactivity period, the scheduler will insert it at the rear of lowQ if $cr_f \leq 0$ and at the rear of highQ otherwise.

Algorithm 1 Tandem Queue (TQ)

```
Init: ∀f, cr_f = 0;
Select next flow:
f = null; selectedLowQ = false;
if highQ.empty = true then
   if lowQ.empty = false then
      f = lowQ.dequeue();
      selectedLowQ = true;
   end if
else
   f = highQ.dequeue();
end if
if f ≠ null then
   cr_f = cr_f - L;
   if selectedLowQ then
      cr_f = cr_f + w_f;
   end if
   Serve L units from flow f
   Reprogram flow (f)
end if
Reprogram flow (g):
if g is eligible then
   if cr_g > 0; then
      highQ.enqueue(g);
   else
      lowQ.enqueue(g);
   end if
end if
```

Note that by constraining $w_f \geq L_{\max}$, we prevent a flow's credit from dropping too low[2], hence $cr_f \in (-L_{\max}, w_f)$. Note also that if we used only one control queue, then the service would be RR: after being served, a flow would be assigned the lowest priority among all other eligible flows. Obviously, such a scheme will not be fair if packets have variable size; additionally, flow weights will not have any effect. By having a lowQ and a highQ, our algorithm maintains *weighted* fairness, independently of the per-flow packet-size distributions. This is described next.

Consider a time interval (t_1, t_2) during which flows $(f_1, \cdots, f_N)$ are continuously active, i.e., eligible for service. In addition, assume that all flows start with $cr_f = 0$ and are initially in lowQ, and that all packets in the system are L bytes long. Assume that initially f_i is at the head of lowQ, and thus the scheduler selects it first and serves its HOL packet. The scheduler will then increment cr_{f_i} by $(w_{f_i} - L)$. The updated cr_{f_i} will be positive if $w_{f_i} > L$, in which case the scheduler will enqueue f_i in highQ. Being alone in highQ, f_i will be selected another $(k_{f_i} - 1)$ times, sending one packet each time, where $k_{f_i} = \lceil \frac{w_{f_i}}{L} \rceil$. Eventually, $cr_{f_i} = w_{f_i} - k_{f_i} \cdot L \leq 0$, and the scheduler will enqueue f_i at the rear of lowQ. Next, the scheduler will select the new head of lowQ, f_j, and will similarly serve it for k_{f_j} times until $cr_{f_j} = w_{f_j} - k_{f_j} \cdot L \leq 0$.

Let a *visit* include the entire service assigned to flow f contiguously in time[3]; also denote by *round* a segment in the execution of the algorithm that visits each active flow exactly once. From the discussion above, it then follows that, in the first round, each flow received service roughly proportional to its weight. If a flow received more service (e.g. 70) than what its weight allows (50), it will end up with a negative credit (-20), which will be accounted for by giving the flow less service in the next round. Thus, if we denote by $s_f(m)$ the service given to an active flow f when the arbiter visits it for the mth time, and $cr_f(m)$ its credit after the mth visit, then it is easy to see that $s_f(m) = w_f + cr_f(m-1) - cr_f(m)$. By letting $S(t_1, t_2) = \sum_{i=1}^{m} s_f(i)$, noting that $-L_{\max} < cr_f < w_f$, and by evaluating the sum, we immediately obtain the following result, which is analogous to Lemma 2 in [15]:

Lemma 1: In any interval (t_1, t_2) during which all flows in the system are active, and flow f is visited m times, the aggregate service assigned to f will be $(m-1) \cdot w_f - L_{max} < S_f(t_1, t_2) < (m+1) \cdot w_f + L_{max}$.

The TQ scheme differs from DRR because for a set of simultaneously active flows with positive credits, TQ will serve one packet from each flow in a RR fashion. In contrast, DRR will serve the flow selected in a burst until its credit has been exhausted, or the next packet is larger than the remaining credit. DRR avoids serving a packet when this is not accommodated by the current credit of the flow; this might leave a flow with up to $L_{max} - 1$ credits (surplus) for the next round. Our scheme, in contrast, serves a flow even if the flow's credit is not sufficient for its next packet, and this can create a debit (negative credit) of magnitude up to $L_{\max} - 1$. Nevertheless, for a time interval in which the set of active flows does not change, our algorithm has similar fairness properties as DRR.

Given that the eligibility of flows does not change, the highQ will always contain at most one flow; this can easily be inferred from the algorithm description. It

[2]Negative credits are also used in [18].

[3]We ignore here the trivial case where f is the only active flow and thus receives all service.

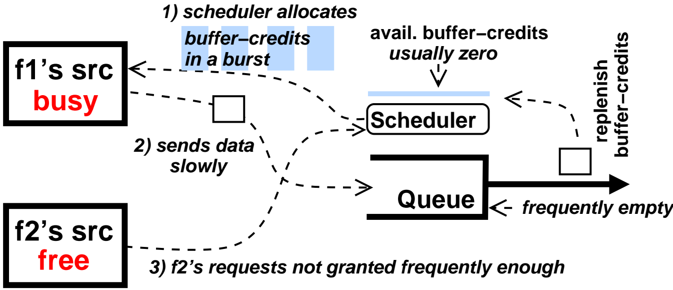


Fig. 2. A scheduler allocating buffer space of an output queue to requesting sources. Buffer credits get replenished when data is forwarded out of the queue.

then also follows that the algorithm visits flows in a RR fashion, much like in DRR: between any two visits to flow f_i, all other active flows will have received one visit. Thus, all active flows are visited the same number of times, plus-minus one. Then, from Lemma 1, it follows that, in the long run, the service given to each flow will be proportional to that flow's weight, independent of the packet size distributions of the flows.

B. Improving the short-term fairness of TQ scheduling

As shown in [15], in any execution of DRR, and for any pair of flows that are active in (t_1, t_2): $S_{f_1}(t_1, t_2)/w_{f_1} - S_{f_2}(t_1, t_2)/w_{f_2} \leq c$, where c is a constant with respect to the number of flows and depends on $L_{\max}$. The same inequality also holds true for TQ if the set of active flows does not change in (t_1, t_2). The above sets a constant upper bound for the difference between the normalized service rates received by any flow pair. (The ideal, fluid GPS scheduler provides continuously equal normalized service rates to active flows [13].) However, in practice, constant c may be large enough to hurt performance.

Consider for instance that we configure three flows, f_1, f_2, and g, with weights $w_{f_1} = w_{f_2} = 100 \cdot L_{\max}$ and $w_g = L_{\max}$. The intention is to treat flows f_1 and f_2 equally, but if one of them competes with flow g, it should get $100\times$ more service. If f_1 and f_2 are active while g is not, then both TQ and DRR will serve $\approx (100 \cdot L_{\max})$ bytes from each flow in turn, thus leading to burstiness. To make things worse, a large MTU may elicit an increase of the weights and therefore of service burstiness.

Burstiness is traditionally undesired in packet-switched networks. In Fig. 2, we depict a scheduler that allocates buffers of a (flow-controlled) destination queue. The scheduler first visits flow f_1, because f_2 presented its requests a couple of clock cycles late. If the scheduler is too bursty, then, before f_2 gets a first chance, there may be no more buffers available. Effectively, f_2 must wait for f_1 to replenish some buffers. If f_1 does not forward the granted data fast enough, then the output queue can underflow, and the output line can stay idle even though it could instead serve f_2's packets.

We now describe a second algorithm, *Tandem Queue Smooth (TQ-Smooth)*, which improves the smoothness (or short-term fairness) of TQ in such practical cases when flows have approximately equal but relatively large weights. The main idea is to prevent one flow from monopolizing the highQ for a long time.

In particular, assume that flow f is now served from lowQ, and is enqueued in highQ. At this point, if lowQ is non-empty, we set the variable *avoidHighQ*. While avoidHighQ remains true, the scheduler continues to visit new flows from lowQ. Once lowQ drains out, the scheduler serves the flows enqueued on the highQ in a RR fashion, at one packet per visit. TQ-Smooth shares many parts with TQ. Algorithm 2, below, depicts their differences.

Algorithm 2 TQ-Smooth: modifications to TQ.

```
Init: avoidHighQ= false;
Select next flow:
selectedLowQ = false;
if highQ.empty = true ∨ avoidHighQ = true then
   f = lowQ.dequeue();
   selectedLowQ = true;
else
   f = highQ.dequeue();
end if
if selectedLowQ ∧ lowQ.empty = false then
   avoidHighQ = true
else
   avoidHighQ = false;
end if
```

The key to understanding why TQ-Smooth provides weight-proportional fair service is the following. A flow that is selected from the lowQ will enter the highQ, re-enter it a number of times, and finally return to the lowQ. In such a time period, the flow f is scheduled a number of times that is proportional to its weight w_f and inversely proportional to the amount of service it receives for one scheduling operation (e.g. the packet size). At the end of this period, the flow drops off the highQ. This results in denser visits to flows with higher weight or smaller packets, which will remain on highQ longer.

We can formally prove this by introducing the notion of the *super-round*. Assuming that all flows are continuously active, a super-round begins with all of

them in lowQ, includes the visit to one or more flows, and ends up the next time instant when all flows fall into lowQ again. Observe that in each super-round every active flow is visited at least twice: once from the front of lowQ and once from the front of highQ. In terms of aggregate service, a super-round is equivalent to a round in the original algorithm or DRR. In particular, if $s^*_f(M)$ is the service of f during the Mth super-round and $cr_f(M)$ is f's credit at the end of it, then $s^*_f(M) = w_f + cr_f(M-1) - cr_f(M)$. The following Lemma can be proven by taking the sum $\sum_{n=1}^{M} s^*_f(n)$, and noting that $-L_{\max} < cr_f < w_f$.

Lemma 2: In any interval (t_1, t_2) *that comprises exactly* M *super-rounds, the aggregate service assigned to* f *will be* $(M-1) \cdot w_f - L_{max} < S_f(t_1, t_2) < (M+1) \cdot w_f + L_{max}$.

III. Evaluation

In this section we evaluate TQ and TQ-Smooth using computer simulations. We do not present the results for DRR separately as they match closely with those of TQ.

In our experiments, we configured a scheduler to arbitrate the access on a 100G link. The flows that compete for the link queue up their packets in front of the scheduler. In every iteration, the scheduler selects a flow and serves its HOL packet; therefore, the scheduler decrements cr by the exact amount of service assigned to the flow selected. We set $L_{\max}$ equal to the maximum transfer unit (MTU) = 1500B. The duration of an $L_{\max}$ packet on the link is 120 ns.

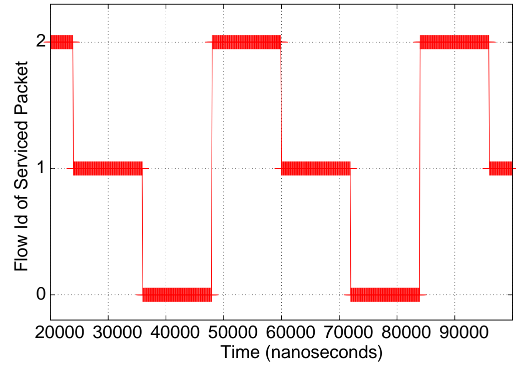
We ran the following experiments.

- **Exp1:** Three persistent flows, all having a weight $w = 100 \cdot L_{\max}$ and sending $L_{\max}$ packets.
- **Exp2:** Three persistent flows, all having a weight $w = 100 \cdot L_{\max}$. Flows 0, 1, and 2 send 1500B, 512B, and 64B packets, respectively.
- **Exp3:** Same as Exp1, but for ten (10) persistent flows.

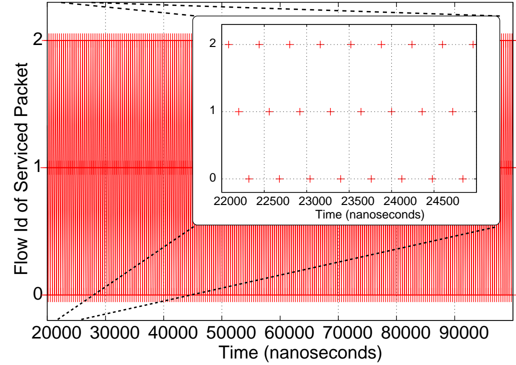
In all experiments, all algorithms assigned to flows equal shares of the 100G link.

In Figs. 3(a) and (b), we present the service time-series of TQ and TQ for Exp1. As can be seen, TQ serves flows in bursts of w bytes, each one lasting for 100 120 ns. On the other hand, TQ-Smooth provides much smoother service, serving one packet at the time in a RR fashion as shown by the inlet of Fig. 3(b).

Figures 4(a) and (b) present our results for Exp2. In this experiment, flow 0 sends larger packets than flow 1, which sends larger packets than flow f2. As can be seen in the figure, TQ again serves flows in bursts of w bytes. On the other hand, TQ-Smooth frequently round-robins,



(a) Exp1: TQ/DRR.



(b) Exp1: TQ-Smooth.

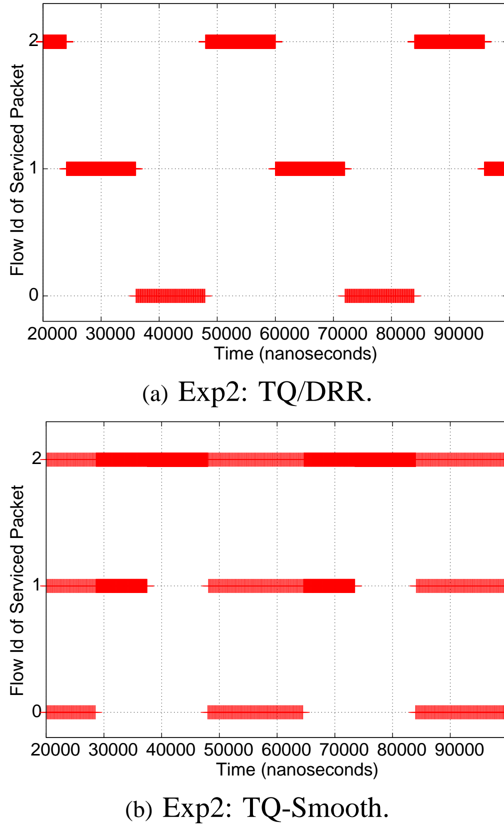
Fig. 3. Results for experiment 1.

sending one packet from each flow in turn. This is when all flows are active in highQ. When flow 0 drops off the highQ, the visits to flows 1 and 2 become denser. Next, the service credit of flow 1 becomes negative, which leaves flow 2 alone in highQ to receive its fair share, until it also drops into the lowQ, thus ending a super-round.

The results of TQ and TQ-Smooth for Exp3 are presented in Figs. 5 (a) and (b). As can be seen, TQ-Smooth serves one packet at a time visiting a different flow in each iteration. This validates the smoothness of the algorithm regardless of the number of flows. In Fig. 5, the horizontal axis measures time in 1500B frame durations.

IV. Optimized buffer credit allocation

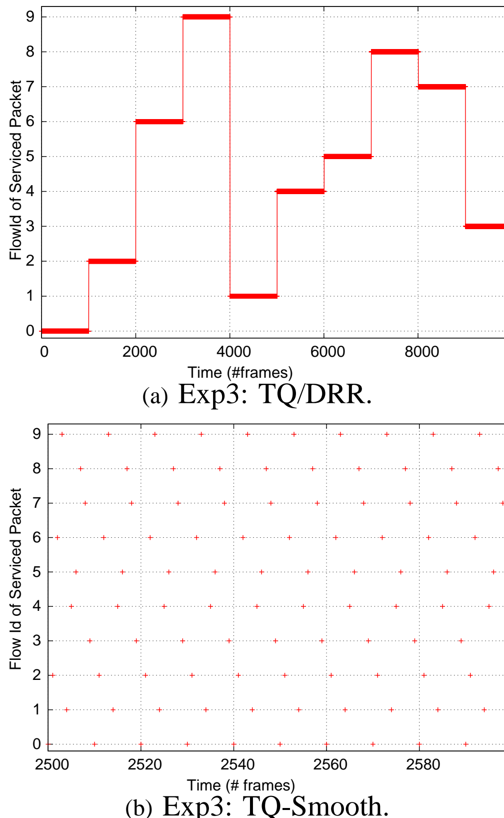
As outlined in Sec. I and Fig. 1(b), large switching fabrics strongly benefit from end-to-end flow and congestion control schemes. In these settings, a scheduler is used at every egress port to allocate egress buffer space to the requesting sources. This simple proactive scheme can prevent overloaded egress buffers (and their accompanying saturation trees) and eliminate deadlocks in the reorder/reassembly buffers [9] [10].



(a) Exp2: TQ/DRR.

(b) Exp2: TQ-Smooth.

Fig. 4. Results for experiment 2.



(a) Exp3: TQ/DRR.

(b) Exp3: TQ-Smooth.

Fig. 5. Results for experiment 3; note that time measures in 1500B transfer-time units.

The operation of a credit scheduler at a fabric-egress port is depicted in more detail in Fig. 6. To inject a packet of size s, a source must first post a request and receive a grant for $g = \lceil \frac{s}{b} \rceil$ buffer units from the corresponding egress scheduler. Although s can be greater than b, for simplicity we assume that $s \leq b$, so $g = 1$. The scheduler stores the outstanding requests in per-flow request counters that are held in on-chip arrays. Each requested or granted credit is for one buffer unit, even when the requesting packet is of smaller size. The scheduler also maintains the overall number of available buffer units (e.g., for one 802.1q priority) in a buffer-credit counter, consuming one buffer credit for each grant provided to a flow, and replenishing one for each packet that departs from the egress queue.

The credit scheduler considered here selects the next flow using TQ-Smooth[4]. The service credit, cr, of each flow counts buffer units, and $L_{\max}$ is the maximum number of buffer units that the scheduler allocates in one shot to a flow. For simplicity, we assume that $L_{\max} = 1$, noting that in practice $L_{\max} > 1$, e.g., to reduce the bandwidth overhead of grant messages. The weight of each flow must be greater than or equal to $L_{\max}$, i.e., $w_f \geq 1$, and corresponds to the number of grants that we want to allocate to a flow in one super-round.

A. *Long-versus-short packet fairness*

At the application layer, high-bandwidth flows mainly comprise MTU packets. Short packets are primarly used for synchronization and other control functions, such as barriers, remote read requests, cache invalidations, TCP acks, flooding network addresses, etc. At an aggregation point in the network, where multiple application layer flows converge, short packets may constitute a significant fraction of the traffic and handling them fairly is important for obtaining low upper-layer protocol latencies.

In this section, we present our solution to allow the network designer to select the most appropriate buffer unit size based on design trade-offs, without having to worry about excessively punishing short-packet flows.

In practice, the buffer memory cannot be divided into arbitrarily small units so as to satisfy each extra byte of payload in a non-overprovisioned fashion. Instead, it is typically divided into fixed-size slots (buffer units), whose size is a trade-off between memory utilization and bookkeeping overhead. The larger the buffer units, the smaller the size and the number of pointers that link them together; on the other hand, too large a buffer unit

[4]Although any other DRR derivative would work as well, as explained in Sec. II-B and Fig. 2, the smooth service of TQ-Smooth is highly desirable when allocating buffers.

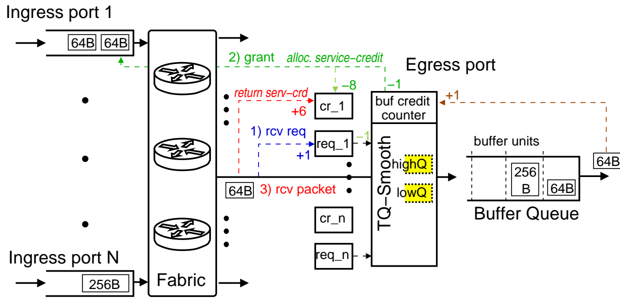


Fig. 6. A scheduler at an egress port of a switching fabric that allocates buffer units in the corresponding egress queue to the requesting ingress ports. Note that the figure omits the service-credit increment by w_f. The numbers shown here are for $b = 256$B, $k = 32$B, and $L_{\max} = 1$. In our experiments, optimal performance is achieved for $k = 4$B.

will result in severe underutilization of memory when the workload includes many small packets. In the remainder of the paper, without loss of generality, we consider buffer units of size b equal to 256 bytes.

Now consider two ingress ports sourcing 64- and 256-byte packets, respectively, which target the same fabric-egress port. Although the corresponding credit scheduler may grant the two sources equally, the 256B one will achieve four times more [bytes/sec] throughput. Note that this problem can be solved if we resort to 64B buffer units, as the 256B flow would then need four 64B grants to inject a packet; but fairness could be again compromised under a mix of 65B and 128B packets.

The solution to this problem given in [9] is to allow the source to inject a *multipacket segment* for each grant that it receives. Each such segment may comprise multiple small packets or even fragments from larger packets that belong to the same flow. In this way, a short-packet flow can fully utilize the service it is assigned by the output credit scheduler. However, this method requires considerable extra bookkeeping per buffer-unit in order to maintain the payload boundaries (and the headers) from multiple variable-size packets.

Here we present an alternative solution that achieves long-vs-short packet fairness as shown in Fig. 6. We logically divide each buffer unit into $\frac{b}{k}$ *buffer subunits*, of k bytes each, and let the service credit (cr) keep track of subunits. Correspondingly, $L_{\max}$, and the weights of flows, w_f, must all be scaled by a factor of $\frac{b}{k}$. Thus decreasing k increases the dynamic range of weights and of service credit variables.

We first consider that $k = 32$B, and scale up $L_{\max}$ and w_f by a factor of 8. (Note, however, that considerably better performance can be achieved for $k \leq 8$B.) The scheduler provides a grant as described before, decrementing the buffer-credit count and the counter of outstanding credit requests by one, but now it debits 8 (32B) subunits to the selected flow f, decrementing cr_f by 8.

Observe that the scheduler is still oblivious of the size of individual packets. All it sees is the cumulative request count from each flow, where each request is for a buffer unit. Thus all that it knows is that each injected packet will fit into a buffer unit[5].

The first time that the scheduler encounters the granted packets (and learns their size) is when the latter reach the egress port. In our scheme, when an egress port receives a granted packet, it informs its scheduler about the *unused subunits* $= \lfloor\frac{(b-s)}{k}\rfloor$, which banks them in the corresponding cr_f counter, thus returning the unused subunits that were debited to its account when the grant was issued to the flow. Hence, as will be shown in more detail below, the cr_f counter is effectively decremented by $\lceil\frac{s}{k}\rceil$ subunits for a grant of 1 credit followed by the reception of packet of size s.

B. Experiments

We conducted computer simulations to test the proposed scheme. In our tests, two flows request credits at full speed from an egress credit scheduler, which grants $L_{\max} = 1$ (buffer-unit) credits at a time.

Test1 (baseline test): Here we did not use the proposed optimization, or, equivalently, the buffer subunits were equal in size to b. One flow was sending 256B, and the other 64B, with b equal to 256B. Both flows had weights equal to 100. As expected, the 256B flow captured 80% of the egress link.

Test2: Next, we set the buffer subunit to $k = 32$B, and accordingly scaled up the flow weights by a factor of 8, setting them to 800. Now, upon granting a buffer unit to flow f, the scheduler decreases cr_f by $\frac{s}{k} = 8$, and when the egress port receives a packet of size $s < 256$B, it increases cr_f by $\lfloor\frac{(256-s)}{32}\rfloor$. Thus, for each packet that the 64B flow sends, its service credit is first decremented by 8 and later incremented by 6, for a net "debit" of 2. The *cost per byte* (C_{IB}) of this flow is thus 1/32. On the other hand, for each packet that the 256B flow sends, its service credit is only decremented by 8, yielding the same cost per byte for either flow.

[5]Note that instead of per-flow request counters we could maintain per-flow request queues to store the size of each individual request. However for large port numbers, this adds significant cost to the implementation.

Remember that when the served flow resides in lowQ, its service credit gets incremented by its weight. Therefore, because the flows have the same $C_{1\text{B}}$, they will receive equal bandwidth shares, which was validated by our simulations.

Test3: Here we repeat test2, but with 64B and 80B flows. As in test2, each packet costs the 64B flow two service credits; in contrast, it costs the 80B flow $8-\lfloor\frac{(256-80)}{32}\rfloor = 3$ service credits. Thus, the $C_{1\text{B}}$ is 2/64 for the 64B flow and 3/80 for the 80B one. Let S_f denote the bandwidth of flow f normalized to the link capacity. As the two flows have the same weight, we expect that $\frac{S_{80\text{B}}}{S_{64\text{B}}} = \frac{C_{1\text{B}}(64\text{B})}{C_{1\text{B}}(80\text{B})} = 0.833$, and as the scheduler is work conserving, $S_{80\text{B}} + S_{64\text{B}} = 1$. It follows that $S_{64\text{B}} = 0.545$ and $S_{80\text{B}} = 0.455$. These rates were validated in our simulations.

C. Variance of cost per byte

In general, the $C_{1\text{B}}$ of a flow sending packets of size $s \le b$ equals $\frac{\frac{b}{k}-\lfloor\frac{(b-s)}{k}\rfloor}{s}$. As b will, in practice, be an integer multiple of k, we have that

$$C_{1\text{B}} = \frac{-\lfloor -\frac{s}{k}\rfloor}{s} = \frac{\lceil\frac{s}{k}\rceil}{s}. \quad (1)$$

Ideally, the $C_{1\text{B}}$ should be the same for all flows, which would allow the scheduler to assign perfectly fair bandwidths. However this is not possible because of quantization effects. It is easy to see that the $C_{1\text{B}}$ reaches its minimum $C_{1\text{B}_{\text{min}}} = \frac{1}{k}$ for packets with size $s = n \cdot k$, $n \in N^+$. Next, we consider some additional packet sizes to find when the $C_{1\text{B}}$ is maximized, and thus to quantify unfairness.

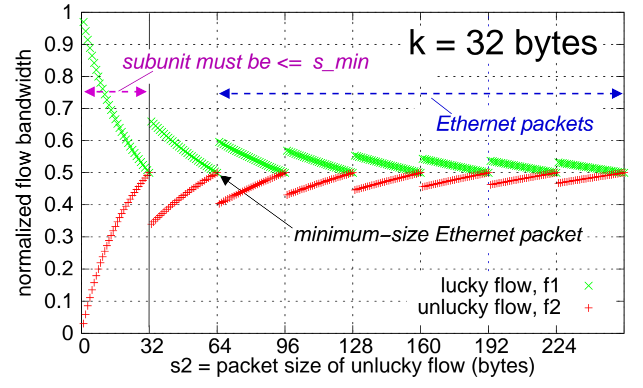
- $s = n \cdot k + k - 1$, Eq. 1 $\Rightarrow C_{1\text{B}} = \frac{n+1}{n \cdot k+k-1} > \frac{1}{k}$
- $s = n \cdot k + 1 \rightarrow$, Eq. 1 $\Rightarrow C_{1\text{B}} = \frac{n+1}{n \cdot k+1} > \frac{n+1}{n \cdot k+k-1}$

It can be seen that the $C_{1\text{B}}$ hits local maxima for $s = n \cdot k + 1$, i.e., for packet sizes that exceed integer multiples of the buffer subunit size by one byte. In addition, the magnitudes of these local maxima do not depend on b, and are decreasing with n. Unless $k = 1$, where $C_{1\text{B}} = 1$, $C_{1\text{B}}$ is smaller for $s_2 = (n+1) \cdot k + 1$ than for $s_1 = n \cdot k + 1$. Note, however, that the $C_{1\text{B}}$ is not a strictly decreasing function of s. For instance, the $C_{1\text{B}}$ for $s_1 = n \cdot k + 1$ is *greater* than that for $s_0 = n \cdot k$.

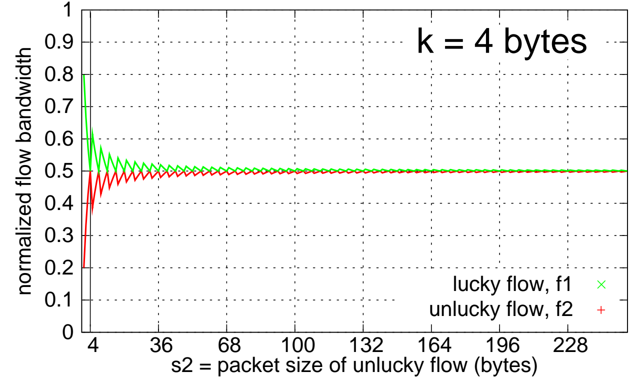
The global maximum of $C_{1\text{B}_{\text{max}}}$ is 1, and is realized with $s = 1$B packets. For $s \in [1, k]$, Eq. 1 yields $C_{1\text{B}} = \frac{1}{s}$. Hence, with packet sizes in this regime, one flow may get nearly k times more bandwidth than another flow. The conclusion is that *we must set the buffer subunits smaller than* s_{min}.

Using Eq. 1, for $s = n \cdot k + 1$, which maximizes C_{1B}, and for $k \le s_{\text{min}}$

$$C_{1\text{B}_{\text{max}}}^{k \le s_{\text{min}}} = \frac{n+1}{n \cdot k + 1} \le \frac{2}{k+1} \quad (2)$$



(a) Approximate fairness for buffer subunit size, $k = 32$B.



(b) Ideal fairness for buffer subunit size, $k = 4$B.

Fig. 7. Normalized service rates of two competing flows. The lucky flow sends packets with sizes $n_1 \cdot k$, thus not losing any service credit. Along the horizontal axis, we vary the packet size of the unlucky flow, s_2. We repeated the same experiments for different packet sizes $s_1 = n_1 \cdot k$ for the lucky flow and the results were virtually identical to those presented here. In Ethernet networks, $s_1, s_2 \ge 64$B.

Therefore, the $C_{1\text{B}_{\text{max}}}^{k \le s_{\text{min}}}$ decreases with increasing subunit size. However $C_{1\text{B}_{\text{min}}} = \frac{1}{k}$ does so as well, and the ratio $\frac{C_{1\text{B}_{\text{max}}}^{k \le s_{\text{min}}}}{C_{1\text{B}_{\text{min}}}} \le 2 \cdot \frac{k}{k+1} < 2$. Therefore, we have an upper bound for the unfairness:

For any $k \le s_{min}$, no flow can grab more than twice the bandwidth of an equal weight flow, regardless of the size of packets.

As we will show later, fairness improves for realistic packet sizes, approaching the ideal for $k \le 8$.

In Fig. 7(a), we configured two competing flows for $b = 256$B and $k = 32$B. The "lucky" flow, f_1, is sending $s_1 = 64$B packets. Along the horizontal axis, we vary the size of the unlucky flow from 1 to 256B, using 1-byte increments. The figure plots the bandwidths of the two flows normalized to the link capacity.

The results validate our previous analysis. As we discussed above, the unfairness is maximized for $s_2 = 1B$,

when the lucky flow gets a nearly 32 times higher bandwidth than $f2$, and decreases as s_2 moves to the right towards 32B. The network designer can avoid this gross unfairness by selecting $k \leq s_{\text{min}}$.

For $s_2 = n \cdot 32$B, the two flows achieve equal bandwidths *regardless of how large we set the packet size of the lucky flow*. For $s_2 = 33$B, the lucky flow gets approximately twice the bandwidth of the unlucky one. These are the worst-case unfair bandwidths that we computed for $s_{\text{min}} \geq k$ using Eq. 2.

Figure 7(a) also shows that for Ethernet packets, $s^{\text{ether}} \geq 64$B, and for $k = s^{\text{ether}}_{\text{min}}/2 = 32$B, the worst-case bandwidth ratio drops to 3/2: the normalized bandwidths are 0.6 (lucky) and 0.4 (unlucky) for $s_2 = 65$B.

Ideal fairness: One drawback of setting k too low is that it increases the dynamic range of flows' weight and service credit variables. But as we discuss below, the benefits can offset the cost.

For practical packet sizes, we can achieve virtually perfect fairness by using appropriately small buffer subunits. Using Eq. 2, for $k = 32$B the worst-case C_{1B} ratio is 1.93, 1.77 for $k = 8$B, and 1.6 for $k = 4$B. Note however that for $k = 8$ and 4, these worst-case ratios assume unrealistically small packet sizes, $s = 5$B and 9B, respectively.

For $k = 8$, and Ethernet packet sizes $s \geq 64$B, the integer n in Eq. 2 will be ≥ 8. Therefore we can obtain a better lower bound $C_{1B}^{k=8,s\geq64} \leq \frac{9}{65}$. Since $C_{\text{1B}_{\text{min}}} = \frac{1}{8}$, the worst-case bandwidths ratio is $\frac{72}{65}$. And for $k = 4$, the corresponding fairness metric becomes $\frac{68}{65} \simeq 1$.

In Fig. 7(b) we repeated the same experiment as in Fig. 7(a) but now for $k = 4$B. As can be seen, for $s_2 = 5$B, the lucky flow gets 1.6 times the bandwidth of the unlucky one. But for realistic packet sizes, e.g., $s \geq 64$B, the two flows achieve virtually the same bandwidths.

V. Conclusions

We presented a practical packet scheduler, TQ-Smooth, that scales to an arbitrarily large number of requestors. Inspired from the success of DRR, the critical path of our algorithm comprises only few low-cost operations, and therefore is readily implementable at ultra-fast link speeds. In terms of efficiency, our algorithm seamlessly integrates smooth service, especially in the common case where flow weights are semi-equal, with weight-proportional fairness. We also described a framework for buffer-credit allocation that uses our algorithms. Finally, we described how to assign equal bandwidths to small- and large-packet flows even when the packet scheduler is unaware of the packet size, and elaborated on the optimal buffer-subunit size.

VI. Acknowledgments

The authors would like to thank Charlotte Bolliger for her kind contributions that improved the quality of the present manuscript.

References

[1] N. Chrysos, F. Neeser, M. Gusat, R. Clauberg, C. Minkenberg, C. Basso, and K. Valk: "Arbitration of Many Thousand Flows at 100G and Beyond", *Proc. Interconnection Network Architecture: On-Chip, Multi-Chip (IMA-OCMC '13)*, Berlin, Germany, Jan. 2013.

[2] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta: "VL2: a scalable and flexible data center network", *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, 2009.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan: "DCTCP: Efficient packet transport for the commoditized data center", *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[4] D. Crisan, A.S. Anghel, R. Birke, C. Minkenberg, M. Gusat: "Short and Fat: TCP Performance in CEE Datacenter Networks", *IEEE Hot Interconnects*, Santa Clara, C.A. Aug. 2010.

[5] F. D. Neeser, N. Chrysos, R. Clauberg, D. Crisan, M. Gusat, C. Minkenberg, K. M. Valk, and C. Basso: "Occupancy Sampling for Terabit CEE Switches", *Proc. IEEE High-Performance Interconnects (HOTI)*, San-Jose, CA, Aug. 2012.

[6] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker: "Deconstructing datacenter packet transport", *Proc. of the 11th ACM Workshop on Hot Topics in Networks*, pp. 133-138. ACM, 2012.

[7] IEEE: "P802.1Qbb/D2.3 - Virtual Bridged Local Area Networks - Amendment: Priority-based Flow Control", 2010.

[8] S. Kavvadias, M. Katevenis , M. Zampetakis , D. Nikolopoulos: "On-chip Communication Synchronization Mechanisms with Cache-Integrated Network Interfaces",*Proc. ACM Intern. Conf. on Computing Frontiers (CF' 10)*, Bertinoro, Italy, 2010.

[9] N. Chrysos and M. Katevenis: "Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics", *Proc. IEEE INFOCOM*, Barcelona, Spain, Apr. 2006.

[10] N. Chrysos: "Congestion Management for Non-Blocking Clos Networks", *Proc. ACM/IEEE ANCS*, Orland, FL, Dec. 2007.

[11] A. Demers, S. Keshav, S. Shenker: "Analysis and simulation of a fair queueing algorithm", *Proc. ACM SIGCOMM*, Austin, TX, Sept. 1989.

[12] M. Katevenis, S. Sidiropoulos, C. Courcoubetis: "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip",*IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, Sept. 1991.

[13] A. K. Parekh and R. G. Gallager: "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks - The Single Node Case", *Proc. IEEE INFOCOM*, Florence, Italy, May 1992.

[14] M. Shreedhar, G. Varghese: "Efficient Fair Queueing Using Deficit Round Robin", *Proc. ACM SIGCOMM*, Cambridge, Massachusetts, Aug. 1995.

[15] M. Shreedhar, G. Varghese: "Efficient Fair Queueing Using Deficit Round Robin", *IEEE/ACM Trans. Netw.*, vol. 4, no. 3. 1996.

[16] L. Lenzini, E. Mingozzi, and G. Steay: "Tradeoffs between Low Complexity, Low Latency, and Fairness with Deficit Round-Robin Schedulers", *IEEE/ACM Trans. Netw.*, vol. 12, no. 4 Aug. 2004.

[17] R. Sriram, P. Joseph: “The Stratified Round Robin Scheduler: Design, Analysis and Implementation”, *IEEE/ACM Trans. Netw.*, vol. 6, no. 6, Dec. 2006.

[18] X. Yuan and Z. Duan: “Fair Round-Robin: A Low Complexity Packet Scheduler with Proportional and Worst-Case Fairness”, *IEEE Trans. Comp.*, vol. 58, Sept. 2008.