Self-Clocked Round-Robin Packet Scheduling

Erfan Sharafzadeh, *Johns Hopkins University and Hewlett Packard Labs;*
Raymond Matson, *University of California Riverside;* Jean Tourrilhes and Puneet Sharma,
Hewlett Packard Labs; Soudeh Ghorbani, *Johns Hopkins University and Meta*

https://www.usenix.org/conference/nsdi25/presentation/sharafzadeh

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Self-Clocked Round-Robin Packet Scheduling

Erfan Sharafzadeh[1,2], Raymond Matson[3], Jean Tourrilhes[2], Puneet Sharma[2] and Soudeh Ghorbani[1,4]

[1]*Johns Hopkins University,* [2]*Hewlett Packard Labs,* [3]*University of California Riverside,* [4]*Meta*

Abstract. Deficit Round Robin (DRR) is the de facto fair packet scheduler in the Internet due to its superior fairness and scalability. We show that DRR can perform poorly due to its assumptions about packet size distributions and traffic bursts. Concretely, DRR performs best if (1) packet size distributions are known in advance; its optimal performance depends on tuning a parameter based on the largest packet, and (2) all bursts are long and create backlogged queues. We show that neither of these assumptions holds in today's Internet: packet size distributions are varied and dynamic, complicating the tuning of DRR. Plus, Internet traffic consists of many short, latency-sensitive flows, creating small bursts. These flows can experience high latency under DRR as it serves a potentially large number of flows in a round-robin fashion.

To address these shortcomings while retaining the fairness and scalability of DRR, we introduce Self-Clocked Round-Robin Scheduling (SCRR), a parameter-less, low-latency, and scalable packet scheduler that boosts short latency-sensitive flows through careful adjustments to their virtual times without violating their fair share guarantees. We evaluate SCRR using theoretical models and a Linux implementation on a physical testbed. Our results demonstrate that while performing on an equal footing with DRR on achieving flow fairness, SCRR reduces the average CPU overhead by 23% compared to DRR with a small quantum while improving the application latency by 71% compared to DRR with a large quantum.

1 Introduction

The Internet is continuously evolving as a multi-party environment. To meet service performance requirements, enforcing isolation at various levels, from tenants to applications and individual flows is crucial [40, 43, 58, 72, 80, 83]. Packet scheduling offers fine-grained control over network bandwidth at both the edge and core of the network. Despite the abundance of scheduling techniques relying on explicit labeling, such as strict priority queues [28, 77, 89] or time wheels [72, 78], obtaining flow priorities or timestamps from Internet transports is usually not feasible. Instead, Fair Queuing provides fine-grained fairness without flow information, and has been deeply studied from fairness, latency bounds, and computational complexity standpoints [21, 32, 41, 42, 64, 68]. Since Fair Queuing schedulers suffer from non-linear computational complexity that hinders deployment [21, 41, 42], a middle-ground alternative is to use Deficit Round-Robin (DRR) scheduling due to its low overhead and scalability [56, 69, 81, 90]. Indeed, DRR is used everywhere from hardware traffic management pipelines to software middleboxes, host stacks, and network interface cards [6, 7, 12, 14, 15, 82, 83].

Regrettably, Internet traffic has evolved since Deficit Round-Robin was proposed, and its drawbacks are increasingly problematic. Workloads like real-time video conferencing and voice communication, streaming, cloud gaming, and instant messaging are now widespread among more traditional web and file transfer [27, 35, 52, 59, 60, 65, 67, 86, 87]. Such workloads feature diverse properties concerning flow counts, flow arrival rates, flow sizes, packet lengths, packet inter-arrivals, and burstiness [22, 23, 27, 39, 67, 85, 86]. First, concerning burstiness, Internet traffic traces show bursts of packets with diverse lengths and inter-arrivals [1, 4, 20, 22, 29]. Long bursts are prone to creating backlogged queues at the packet scheduler, while, broadly, short bursts from latency-sensitive applications such as web and streaming [61, 74] exhibit periodic on-off arrivals. For example, the median response size for web workloads does not exceed 10 kB, while Variable Bit-Rate (VBR) streaming workloads such as Zoom have a median frame length of 10 kB with large idle gaps between frames [61].

Our experiments with DRR reveal that under a backlogged scheduler, a bursty latency-sensitive flow has to wait for up to a full scheduling round, i.e., until all flows have been serviced at least once, before it can send its traffic. This can be detrimental to the performance of such flows, causing much higher application latency compared to Fair Queueing Schedulers. Even with recent DRR enhancements, such as Sparse Flow Optimization (SFO) [47], which prioritize new incoming flows over backlogged ones to improve responsive-

ness, bursts with more than one packet are prone to being quickly demoted to lower-priority and forced to compete with backlogged flows, thus experiencing long delays.

Secondly, packet lengths are diverse and skewed [1,4,16,20, 22,29] and packet length distribution for wide-area network traffic is difficult to generalize. Many modern applications use small packets to reduce latency and the impact of packet loss [30,79]. Network Interface hardware acceleration [11] may produce very large packets (64 kB). DRR relies on a fixed, user-specified quantum equal to (or larger than) the maximum packet size in the network [81], making it hard to tune. A large quantum can be detrimental to intermediate buffers as it allows large bursts of packets to be sent when servicing each flow [56,69], ultimately increasing response times for latency-sensitive applications. *No packet scheduler today offers low resource utilization, low latency, and adaptability to packet sizes without explicit flow information feedback.*

Our work makes several novel contributions to fair scheduling. First, we show that DRR's drawbacks, i.e., high latency for bursty flows and vulnerability to quantum choice, are measurable at the application level, even on high-speed networks. We further identify the trade-off between latency and CPU utilization to be intrinsic to DRR's design and argue that there's no correct answer to configuring DRR (§2).

Second, we outline the design of Self-Clocked Round-Robin (SCRR), a novel multi-queue round robin scheduler. SCRR borrows virtual clocking from Fair Queuing [41,42] to eliminate the quantum and enforce fairness, and services sub-queues like DRR in strict sequence for simplicity and scalability. SCRR is parameter-less, light, scalable, and adapts to bursts and packet length variations. We design various SCRR enhancements to efficiently prioritize bursty flows and reduce application latency. When an idle flow becomes active, the scheduler decides its bandwidth allowance via its virtual clock setting. Backed by strong theoretical virtual clocking principles, SCRR can efficiently adjust the virtual clocks for short, bursty flows, allowing them to quickly make progress without violating fairness bounds (§3). Third, we provide rigorous proofs of flow burstiness and fairness for SCRR, guaranteeing that SCRR has the same fairness as DRR with a low maximum burstiness bound (§4).

Fourth, we evaluate SCRR and competing schedulers on a physical testbed under Cubic and BBRv3 using novel experiments. Application-level fairness is tested with thousands of active flows while application latency is measured for request-response flows and VBR traffic. Our results demonstrate that employing SCRR in a software switch reduces CPU utilization by 23% when compared to DRR with 1500B quantum. Our results further reveal that SCRR offers 93×, 71%, and 45% lower response times over tail-drop, DRR with a small quantum, and DRR with a large quantum, respectively. (§5)

SCRR is a step toward more dynamic packet scheduling for multi-party wide-area deployments. It is the first true alternative to DRR, offering the low latency of Start Time Fair Queuing with the low complexity of DRR. Our ultimate goal is for SCRR to replace existing hardware and software packet schedulers because of its plug-and-play deployment, low resource consumption, low latency, and desirable fairness.[1]

2 Packet Scheduling for Modern Internet

Internet workloads incorporate a mix of heavy, throughput-intensive flows, as well as bursty latency-sensitive flows with diverse arrival patterns. Can Deficit Round Robin [81], as the de facto candidate for fair scheduling cope with modern traffic characteristics?

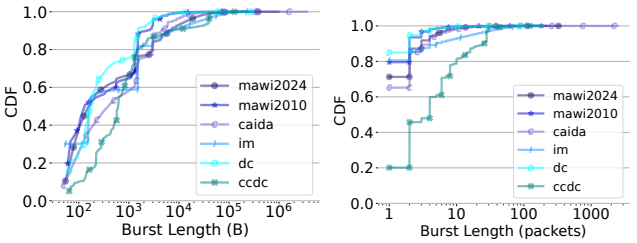
2.1 A brief history of packet scheduling

Historically, Packet-by-packet Generalized Processor Sharing (PGPS) [68] was among the first scheduling paradigms that target fair bandwidth allocation among tenants. Tenants or flows are classified into separate FIFO queues that are then processed by the packet scheduler. Subsequently, numerous Fair Queuing schedulers [32,41,42,45,66,81,90] were introduced with distinct bounds and computational complexities. Classical Fair Queuing algorithms such as Start-Time Fair Queuing (STFQ) [42] have the downside of expensive per-packet sorting operations as they need to constantly track and update the bandwidth allocation for each flow and choose the most eligible sub-queue, i.e., the sub-queue with the least progress. Further, computing requirement scales up with the number of sub-queues.

Deficit Round Robin (DRR) scheduling addresses this shortcoming by allowing constant complexity dequeue operations irrespective of scale [81]. Instead of maintaining a sorted data structure to keep track of flow progress, it visits each sub-queue in a round-robin fashion and allocates a fixed amount of bytes, called *quantum*, to each queue at each round. The part of the quantum that is too small to send the next packet is accumulated as a deficit for the queue. Other variants of DRR, including Weighted Round-Robin (*WRR*) and Modified Deficit Round Robin (*MDRR*) [12] can operate on a limited number of FIFO queues strictly reserved for different flow priorities [6]. These variants have a coarser fairness granularity (per priority group, instead of per-flow) and need to be configured with a deficit value that determines the amount of bytes each priority group transmits on each pass.

Due to its constant computational complexity and low memory footprint, *DRR* is one of the most widely deployed packet schedulers in both software and hardware [6, 7, 12, 14, 15, 81–83]. Even with the emergence of novel queuing abstractions such as *PIFO* [82] and calendar queues [78] that offer advanced traffic classification and shaping, *DRR* endures as a robust and widely used option for flow scheduling, particularly in scenarios where intricate packet

[1] SCRR artifacts are available at https://github.com/jean2/scrr.



(a) *Burst size in bytes* (b) *Packets in a burst*

Figure 1: *Diversity in burst length distributions in the wild.*

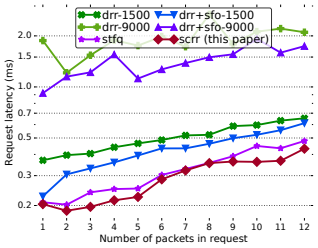


Figure 2: *DRR offers poor latency under different burst sizes.*

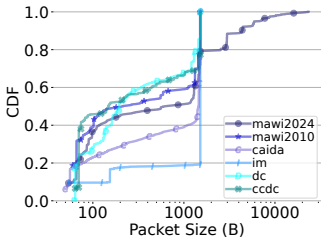


Figure 3: *Skewed packet length distributions in the wild.*

classification is unnecessary [26, 33, 37, 91, 93] and standard TCP support is required.[2]

2.2 Flow burstiness in modern traffic

Burstiness is a well-known phenomenon for Internet traffic due to its multitude of root causes [20, 22, 50, 51, 76, 92]. The emergence of interactive applications such as cloud gaming, video streaming, and video conferencing further underscores a type of traffic that experiences long idle times due to user behavior, network congestion, and variable bit-rate enforcement by the application [20, 44, 61, 75, 87]. Traffic studies further report that many latency-sensitive workloads primarily produce short flows. For example, the median response size for web requests entering Facebook data centers is reported to be less than 10 kilobytes [74], and frames in Zoom video traffic do not exceed 10 kB, with large gaps in between [61].

Fig. 1 presents the CDF of burst length distributions for various Internet traces, ranging from backbone internet (*caida*) [4], campus networks (*dc*) [22], and ISP gateways (*mawi*) [29], to instant messaging (*im*) [20] and anonymized traces (*ccdc*) [1] using an Inter-Packet Gap (IPG) threshold of 50 μs (distributions with other IPG thresholds are similar and reported in Appendix C). The diversity in burst sizes suggests that not all Internet flows tend to remain active for a long time as they arrive at a bottleneck. Notably, many flows exhibit small bursts of packets followed by large idle periods. But we also see a number of very large bursts belonging to heavy flows likely to congest the packet scheduler.

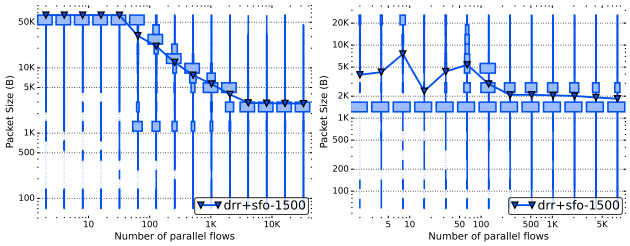
DRR has effectively the same theoretical bounds as Fair Queuing schedulers (§2.1): it is fair and has a maximum service deviation of 2× the quantum [81]. The fairness, burstiness and latency guarantees of *DRR* and Fair Queuing schedulers are based on the assumption that all flows arriving at the scheduler are always backlogged [41, 42, 81]. Alas, the above evidence suggests that this is rarely the case for modern Internet traffic! In practice, DRR usually has higher latency for those short bursty flows which are not backlogged at the scheduler, and this is seldom studied. With gaps in arrivals, short flows are prone to get stuck behind backlogged flows in the scheduler. Fig. 2 compares the response times for a request-response workload under two software implementations of *DRR*, the original design presented in [81], and the state-of-the-art implementation of *DRR* which implements *Sparse Flow Optimization (SFO)* [47], a technique that prioritizes interactive traffic (§3.3.1). We compare *DRR* with two different quantum settings: 1500B and 9000B, and with *STFQ* [42]. Two senders generate a total of 16 parallel streams of requests composed of bursts of 150B MSS packets, while the burst length increases on the x axis. Two other senders generate 64 heavy flows each to ensure the scheduler is backlogged. The full experiment setup can be found in §5.5. Latency gaps as large as 4× between *DRR* and *STFQ* suggest that *DRR* degrades the application performance due to poor scheduling decisions regardless of its quantum setting, as consecutive packets in a burst have to wait for full scheduling rounds before they are transmitted. *DRR+SFO* is initially able to bridge this gap by boosting short flows through the scheduler, but as the burst size increases, consecutive packets of the burst do not benefit from prioritization and have to wait.

2.3 The amplifying impact of packet sizes

Packet lengths are diverse in today's networks. Recent studies report a wide variety of packet length distributions observed in the network [4, 20, 22, 71, 92]. Fig. 3 presents the packet size distribution for a few Internet traces. According to these studies, heavier workloads such as instant messaging (*im*) tend to transmit MTU-sized packets while mixed workloads such as *mawi* send a wider range of packet sizes with a considerable number of short packets in the mix. The available backbone traffic packet traces [4] further highlight that packet length distribution constantly alternates in short periods (e.g., between bursts of 550B, 1500B, and 60B segments). Many modern interactive applications use small packets to reduce latency and to minimize the impact of packet loss, while some even adapt their packet sizes based on performance feedback [30, 79]. The quantum for DRR needs to be set based on the packet size [81], which makes it even harder for operators to tune their fixed quantum scheduler since these variations in packet lengths are not only workload-dependent but also volatile in the short term.

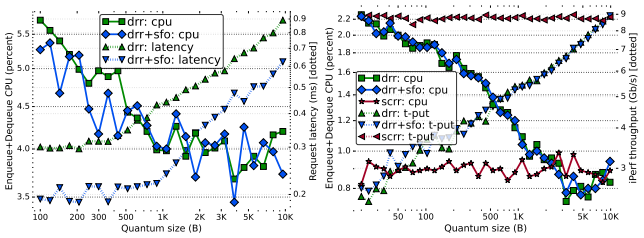
Hardware offloads and large packet buffers make it difficult to configure quanta. Software network stacks rely on

[2] We later compare the performance of state-of-the-art implementations of *PIFO*, such as AIFO [89] and *SP-PIFO* [17], with *DRR* in §5. Our results suggest that *AIFO* suffers from severe throughput loss at large RTTs, while *SP-PIFO* causes heavy packet re-ordering.



(a) *TSO on TX machine* (b) *LRO on RX machine*

Figure 4: *SKB length distribution under NIC offloading functions.*



(a) *CPU vs scheduling latency* (b) *CPU vs app. throughput*

Figure 5: *Trade-offs in DRR's quantum configuration.*

offloading support in the network interface (NIC) to keep up with their increasing bandwidth [25]. On the transmit side, *Transmit Segmentation Offload* (TSO) breaks large Socket Buffers (SKBs) into MTU-sized packets in the NIC [11]. On the receiving side, *Large Receive Offload* (LRO) aggregates small segments into larger frames [11]. In Linux, by default, the SKB size can run as high as 64 kB and with proposals to greatly increase this maximum [34]. The reduction in SKB rate provided by TSO and LRO improves software performance by reducing the overhead of context switches, cache misses, and metadata handling. However, TSO and LRO force any software packet scheduler to deal with a wide and unpredictable range of packet sizes.

For packet schedulers deployed as part of virtual switches in the datacenter or cloud, the packet size distribution will be dictated by *TSO*. Fig. 4a presents the packet size distribution of experiment §5.3, in the form of parallel histograms (wider bins indicate higher occurrence and the straight horizontal line presents the average packet size over the number of parallel flows) as received by the packet scheduler for different numbers of concurrent *Iperf* flows generated on that server. With more flows, TSO tends to generate smaller packets, as each flow has a smaller bandwidth and TSO tries to match the smaller bandwidth delay product.

For packet schedulers that are part of home routers, SD-WAN gateways, access points or middleboxes, the packet size distribution will be dictated by *LRO*. Fig. 4b presents the packet sizes generated by LRO for experiment §5.3. LRO causes less aggregation than TSO, but is also less predictable, likely due to the interplay of LRO with actual packet arrivals at the NIC and NIC interrupt coalescence. TSO and LRO make it harder to set the quantum optimally, and a quantum greater than the largest packet size, 64 kB, is much too large in most cases.

Mis-configuration with Jumbo Ethernet. Many network devices may not be optimally configured, or are configured conservatively. Most end hosts are configured for standard Ethernet, in case a network device does not support Jumbo packets. If a network device enables support for Jumbo Ethernet [57], e.g., in case some end-hosts need it, then DRR on this device would use a 9 kB quantum when the MTU is effectively 1500 B.

2.4 DRR quantum configuration trade-offs

DRR's quantum is usually set to the maximum packet size in the network (MTU) [81]. Such configuration impacts application performance and resource utilization. With a large quantum, *DRR* produces large bursts of packets that belong to a single flow and a latency-sensitive flow has to wait longer to be allocated its share, increasing the application latency (Fig. 2). Conversely, with a quantum smaller than MTU, DRR may need to spend a few scheduling rounds accumulating enough deficit to transmit a large packet, such missed visits ultimately result in wasted compute resources.[3] Fig. 5a presents the results of an experiment with the *DRR* scheduler mixing latency-sensitive 500B requests traffic (a two packet burst) with heavy background traffic using 1500B packets as an attempt to mimic the mixed and diverse nature of Internet bursts arriving at a bottleneck (the experiment is fully described in §5.5). Increasing the quantum to 10 kB increases the latency of the small requests experienced at the application level (by 140% compared to the best-case latency for *DRR+SFO*). Reducing the quantum to 100B raises the CPU utilization of the scheduler to nearly 7% of the system (around 75% increase), in this case 93% of its scheduling visits are missed due to insufficient deficit.

CPU utilization is a good proxy for the overall resource requirements of the packet scheduler. Some software implementations of packet schedulers in Internet gateways result in reduced network throughput due to their high CPU usage [2, 3]. Fig. 5b shows an incast of heavy flows sharing a bottleneck link serviced by *DRR* as we increase the CPU utilization by reducing the quantum. We use *perf* [9] to measure the CPU consumption of the scheduler's qdisc module. At higher quanta, the lower per-packet processing time is balanced by the increasing packet rate, and CPU usage is constant. A small 20B quantum makes *DRR* the bottleneck and reduces aggregate throughput to 3 Gb/s, which is a third of the capacity. During packet processing many parts of the Linux network stack must be exercised (§D.4). The CPU usage for *DRR* is less than 2.5%, however this increases CPU contention, cache misses and overall pressure on the rest of the networking stack, causing this slowdown.

[3] Some hardware implementations of *DRR* use negative deficits, however after sending a large packet, missed visits will be needed to recover the deficit and avoid unfairness. The only way to eliminate missed visits is either to accept unfairness or to restrict the quantum to be greater than MTU.

3 Self Clocked Round Robin

An ideal packet scheduler should adapt to workload dynamics, such as varying packet sizes and burstiness, both at the flow and packet level, without the need for explicit user input. Conversely, DRR requires quantum configuration based on packet lengths and flow types. In addition, a widespread deployment requires scalability, resource efficiency, and tight bounds on both performance and fairness, which cannot be satisfied altogether by Fair Queuing schedulers. In this section, we introduce *Self-Clocked Round-Robin Scheduling (SCRR)* to address those needs. We design SCRR with the following goals in mind:

- Per-flow or per-class QoS scheduling.
- Fair bandwidth allocation.
- Low scheduling latency for bursty flows.
- Low resource consumption at high loads.
- Scalability to large number of sub-queues.
- Compatibility to any transport, including TCP.
- Parameter-less and agnostic to flow information.
- Plug and play replacement for DRR and Fair Queuing.

To envision SCRR, we find that virtual clocking proves to be a great asset in realizing dynamic scheduling paradigms that adapt to workload changes. Unlike conventional deficit round-robin that sends a predetermined burst on each sub-queue visit, SCRR aims to track virtual times for sub-queues and transmit as much as a flow sub-queue's virtual time allows in order to achieve fair scheduling without relying on any flow information. In order to eliminate the expensive ordered list maintenance as in classical fair scheduling [41, 42], SCRR visits each flow sub-queue in a round-robin fashion. SCRR can be used in both low-load and high-load environments as a parameter-less improvement over DRR. (§3.1)

SCRR is flexible and is augmented by various customizations and enhancements. SCRR can optionally use weight for the sub-queues (§3.1) [32]. It can use the *virtual finish time*, like SCFQ [41], or *virtual start time*, like STFQ [42]. *No Packet Metadata* (§3.2) eliminates the need to tag packets with their virtual time while enqueued, reducing resource requirements. The combination of *Sparse Flow Optimization* (§3.3.1), *Initial Advance* (§3.3.2), and *No Empty* (§3.3.3) use the properties of virtual time to decrease the latency of bursty flows beyond the state of the art, without violating the fairness and burstiness guarantees of SCRR. With these enhancements, when both heavy and short flows are present at the scheduler, bursty flows will not have to wait for multiple scheduling rounds to make progress and enjoy lower scheduling latency.

3.1 Virtual Clocking Semantics

We start by laying out the formal foundations of the SCRR scheduler. All mathematical notations used are summarized in Table 1. The SCRR scheduler maintains a *global virtual clock*, which is updated to the maximum virtual time of dequeued packets at the end of each round. Each sub-queue features a head virtual time which is determined by the virtual time of the packet at the head of the queue. Depending on the head virtual time, sub-queues can be either *lagging* or *leading* compared to the global virtual clock. During each sub-queue visit, the scheduler allows a lagging sub-queue to catch up with the global virtual scheduling time, while a leading sub-queue advances the global virtual clock. During a period that is longer than a scheduling round, sub-queues may change their state from *lagging* to *leading* and vice versa.

Definition 1. A *scheduling round* is the period of time when all active sub-queues have been visited once, and only once. A sub-queue is active when it contains outstanding packets.

Definition 2. A sub-queue is *lagging* (catching up) as long as its head virtual time is smaller than the global virtual clock.

Definition 3. A sub-queue is *leading* as long as its head virtual time is larger or equal to the global virtual.

During each round, the scheduler visits sub-queues in round-robin and forwards all packets that have a virtual time that is older than the global virtual clock. If a sub-queue does not feature a lagging packet, the head packet is transmitted, and the scheduler moves to the next sub-queue. This ensures that no processing cycle is wasted when visiting each sub-queue, while still ensuring that fairness can be achieved via virtual clocking. SCRR effectively implements round-robin scheduling with byte-level fairness granularity and an adaptive quantum equal to the largest serviced packet in a round.

Enqueue Process. Since SCRR is a multi-queue scheduler, the arrived packets undergo a classifier to determine their target sub-queues. In implementations that target per-flow fairness, each flow is stochastically assigned to a separate sub-queue using a flow classifier (e.g., *fq* in Linux). If the sub-queue is backlogged, the virtual time of the tail packet plus its size is used as the virtual time for the new packet. However, if the sub-queue had been idle, the current global clock is used as the packet virtual time. This is to ensure that an idle sub-queue does not receive a large (and consequently, unfair) scheduling share due to inactivity for a long period.

Formally, at scheduling round k, the virtual time of packet p_f^j, i.e., the j'th packet of sub-queue f, is defined as:

$$S(p_i^j) = max\{F(p_i^{j-1}), c(k)\} \quad (1)$$

where $F(p_i^j)$ is the virtual finish time of the previous packet and $c(k)$ denotes the current virtual time of the scheduler. The virtual finish time for packet p_i^j is defined as:

$$F(p_i^j) = S(p_i^j) + \frac{l(p_f^j)}{r_f} \quad (2)$$

Table 1: *Description of mathematical notations.*

Notation	Description
p_f^j	Packet j of flow f
$l(p_f^j)$	Length of packet p_f^j
$c(k)$	Global virtual clock at scheduling round k
$v(p_f^j)$	Virtual time of packet (p_f^j)
r_f	Rate (weight) of flow f
N	Number of packets in the scheduler
Q	Vector of active sub-queues in the scheduler
n	Number of active sub-queues in the scheduler
M	Maximum length among all packets

where $l(p_f^j)$ denotes the length of the packet and r_f is the weight of the sub-queue f. Algorithm 1 in Appendix A presents the enqueue procedure.

Dequeue Process. The majority of virtual timekeeping processes take place during the packet dequeue. SCRR starts with the first non-empty sub-queue and schedules one packet. The scheduler then loops through that sub-queue and transmits packets as long as the virtual time of the head packet is older than the global virtual clock. This approach accomplishes two key objectives. First, it guarantees the transmission of at least one packet from each sub-queue, preventing missed schedule opportunities that lead to wasted compute cycles. Second, it helps the lagging sub-queues to catch up with the leading sub-queues by sending multiple packets. Upon moving to the next sub-queue, the scheduler records the start time of the last dequeued packet and cycles through the remaining active sub-queues. When a scheduling round is finished, SCRR updates the global virtual clock to the maximum packet virtual time that was transmitted during that round. Therefore, an update of the global clock only occurs once during each round to ensure the correct operation of the scheduler. The dequeue procedure is outlined in Algorithm 2 in Appendix A.

The self-clocking semantics in SCRR uses Start-Time Fair Queuing (STFQ) [42] as its basis. Alternatively, as a design choice, it is possible to schedule packets using their virtual finish times as in [41]. In §4, we perform all the correctness proofs for the *start time* version, and in Appendix B.1, we do the same for the *finish-time* version. The two self-clocking paradigms offer identical fairness and burstiness bounds for SCRR and differ slightly in the state-keeping required for the scheduler's implementation. It is also worth noting that all virtual time calculations in SCRR support sub-queue weights to implement QoS, i.e., the packet's length is divided by the weight of its corresponding sub-queue.

3.2 Eliminating per-packet metadata

Both SCFQ [41] and STFQ [42] need to store the virtual time of the packet while the packet is enqueued. Since this is usually done in the packet metadata, it has the downsides of added complexity and increased memory operations. **No Packet Metadata** (NPM) is an enhancement to the SCRR algorithm that computes packet metadata after dequeuing the packet, and therefore eliminates the need to store the virtual time of the packet as metadata.

When dequeuing a packet, the first issue is to know if the sub-queue was idle or backlogged when the packet was enqueued (§3.1). With **NPM**, the scheduler saves the global virtual clock of the previous round. If the virtual time of the last dequeued packet in the sub-queue is greater than previous virtual clock, then that sub-queue was scheduled in the previous or current round, and was backlogged. When a sub-queue is backlogged, the start time of any queued packet can be derived from the finish time of its predecessor in the sub-queue. Consequently, the sub-queue must store the finish time of the last dequeued packet, instead of the finish time of the last enqueued packet. When a sub-queue was idle, NPM always sets to virtual time of the packet to the current global clock. When the sub-queue was idle, the packet virtual time is supposed to be the value of the global virtual clock at the time the packet was enqueued (Equation 1). The insight is that in SCRR, precise virtual timing does not really matter as long as the packet is included in the proper scheduling round and its virtual time does not change the computation of the next global clock. Further, in many cases the sub-queue was added to the schedule in the current round, so the packet virtual time would have been the current global clock anyway.

NPM computes the virtual time of dequeued packets based on the scheduler's previous global time and the previous packet's finish time in the same sub-queue. This simple heuristic makes SCRR as memory efficient as DRR while making the enqueue process a breeze by removing four memory operations during insertion. The theoretical proofs obtained for SCRR also apply to SCRR with NPM, using either the finish time of packets or their start times.

3.3 Flow Latency Enhancements for SCRR

Packets of *backlogged flows* need to wait for all prior packets of the sub-queue to be dequeued, so their queuing delay is the sub-queue's length times the duration of each scheduling round. The latency of such packets is dominated by the queuing delay, therefore their exact position in the scheduling round has little impact on their latency. Further, backlogged flows are usually associated with throughput-intensive applications whose performance is dictated by the fairness of the scheduler. For backlogged flows, round robin schedulers, such as DRR and SCRR, are in practice as good as Fair Queuing schedulers, such as SCFQ and STFQ.

Non-backlogged flows, also called *sparse flows*, don't accumulate packets in the sub-queue, therefore, their packets are immediately eligible to be dequeued. The position of those packets in the scheduling round makes a big difference to their latency. Further, interactive and latency-sensitive

applications are major origins of sparse flows, or flows that alternate between sparse and lightly backlogged. Our aim is to make the latency of SCRR for sparse flows as good as the Fair Queuing schedulers.

Our enhancements for SCRR reduce the latency of *sparse flows*, and as a result make the latency of backlogged flows slightly worse than DRR. As pointed above, backlogged flows are almost never sensitive to such minor differences in latency. This design choice aims to improve the latency of flows that care the most about it. This enables to balance the scheduling performance for various types of flows without explicit flow information.[4] The above enhancements involve various trade-offs between resource usage, latency, and fairness. Hence, it is at the discretion of the network operators to decide which combination of SCRR features to use. We empirically evaluate the direct impact of each of these enhancements on application performance in Appendix D.2.

3.3.1 Sparse Flow Optimization

Sparse Flow Optimization (SFO) is a technique developed for DRR that reduces the latency of bursty flows by prioritizing sparse flows [47]. **SFO** was trivially added to SCRR. In **SFO**, sub-queues that were previously inactive (i.e. sparse) are no longer added at the back of the schedule, but instead in the *new list* of sub-queues that always take precedence over the *old list* of sub-queues (that were already part of the schedule, i.e. backlogged). To prevent sparse sub-queues from starving backlogged sub-queues, newly empty sub-queues are kept in the schedule and are only potentially removed from it during the next scheduling round [19].

3.3.2 Initial Advance for new sub-queues

The trivial version of SFO in *SCRR* does not benefit the bursty flows as well as SFO for *DRR*. With *DRR*, if multiple packets have accumulated prior to a sub-queue being scheduled via SFO, those packets can be sent in the current scheduling round, up to a quantum. With basic SCRR, when a sub-queue has been idle, the virtual time of the first packet is set to the current virtual clock (same as *SCFQ/STFQ*). When the sub-queue is scheduled with SFO, the first packet can be sent, but any subsequent packet has a virtual time in the future and needs to wait for the next scheduling round.

Initial advance fixes this issue by allowing SFO to send multiple packets in the first schedule of a previously inactive sub-queue. It simply sets the virtual time of the first packet based on the previous virtual time instead of the current virtual time (Fig. 6c). The virtual time of subsequent packets is based on the virtual time of the first packet, packet size, and the sub-queue weight, therefore setting a virtual time closer

[4]The Completely Fair Scheduler is a relevant case study. While the original process scheduler design heavily depended upon the use of nice values for latency-sensitive processes, the scheduler was gradually improved to automatically favor interactive applications [24].

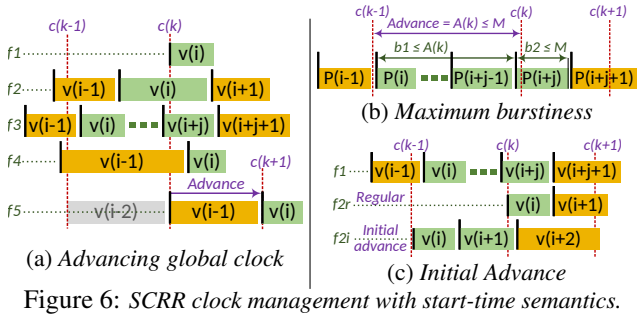


Figure 6: *SCRR clock management with start-time semantics.*

to the previous virtual clock enables potentially more subsequent packets to have a virtual time lower than the current virtual time, and consequently to send more packets in current schedule.

Initial advance will not violate the scheduler's fairness and burstiness properties. The current scheduling round processes packets from sub-queues with virtual times later than the previous virtual clock $c(k-1)$ (Fig. 6a). Therefore, setting the virtual time of the first packet to any value later than the previous virtual clock is still compatible with SCRR. To further prevent unfairness and avoid starvation, SCRR makes sure that the virtual time of a sub-queue used to tag packets can only go forward, not backward. **Initial advance** can be further tuned. To reduce the average flow burstiness, we set the virtual time of the head packet to the previous virtual clock plus the packet's size times the sub-queue weight: if all packets in the sub-queue are the same size, the burst of packets from the sub-queue scheduled by SFO is equal to the advance, instead of the advance plus an additional packet (Fig. 6b).

3.3.3 Getting rid of empty visits with No Empty

SFO in DRR keeps newly empty flows in the schedule for at least one round to prevent unfairness and starvation [19]. This means that the next time that sub-queue is scheduled, if it is still empty, the scheduler needs to remove it from the schedule and pick another sub-queue. This is problematic in software as such empty visits create extraneous cache misses. This is also problematic in hardware with fixed-delay processing pipelines, it makes the amount of work to schedule one packet unpredictable, as many successive empty sub-queues may need to be processed.

No Empty improves SFO and eliminates these empty visits by leveraging the virtual clocking semantics. With **No Empty**, idle sub-queues are never kept in the schedule. To prevent starvation, when a previously inactive sub-queue needs to be inserted into the schedule, its virtual time is checked against the current clock. If a sub-queue has transmitted more than its fair share of packets in the current schedule, its virtual time will have advanced beyond the global virtual clock. If a sub-queue has a virtual time greater than the global virtual clock, it is denied the use of SFO, and is instead added to the back of the *old list* of sub-queues, similar to the original DRR.

Otherwise, it can use SFO and is added at the back of the *new list* of sub-queues.

No Empty is especially attractive to hardware implementations as it makes the processing of SCRR with SFO completely deterministic, without loops and recursions. It also combines with **Initial advance** to benefit bursty flows. In many cases, the packets of the burst do not arrive at the same time. With the original SFO, only the packets already accumulated prior to SFO scheduling are sent in the current schedule, if a packet arrives later, the sub-queue is at the back of the schedule and the packet has to wait a full scheduling round. With **No Empty**, when this later packet arrives, the sub-queue is inactive, and can be inserted in the current scheduling round another time, as long as the virtual time of the packet is older than the current virtual clock.

4 Theoretical Analysis

In this section, we explore the theoretical properties of the scheduler and provide upper bounds for the throughput, fairness, and burstiness.

4.1 Analysis of Backlogged SCRR

Initially, we show that the movement of the global clock under SCRR is tightly bounded during each scheduling round. Next, using this insight, we establish the fairness index of the backlogged scheduler and argue that SCRR enjoys the same fairness index as DRR in the long run. Finally, we argue that the per-flow burst sizes are strictly bounded by two maximum-sized packets, similar to DRR.

Definition 4 (**Single Clock Update per Round**)**.** For the smallest time interval $[t_1, t_2]$ that includes a full round-robin scheduling round, i.e., a period when all active flows are visited at least once, SCRR global clock is updated only once according to the scheduled packet with the highest virtual time value.

Definition 5 (**SCRR Global Clock Updates**)**.** All global clock updates in SCRR occur at the end of a scheduling round, the global clock only advances forward, and the clock update marks the start of a new scheduling round.

Theorem 1 (**The Global Clock Advance Upper Bound**)**.** At each scheduling round, the global clock advancement in SCRR, $A(k)$, is bounded by the maximum packet length.

Proof. Suppose that for the SCRR scheduler with backlogged sub-queues, the global clock is set to $c(k)$ at the beginning of a scheduling round. Since SCRR guarantees the dequeue of a packet upon a visit, packet p_f^i is dequeued for flow f. The packet's virtual time (start time) is $v(p_f^i)$. The following cases can be observed (Fig. 6a):

(1) $v(p_f^i) \leq c(k)$:
The dequeued packet has a virtual time earlier than the global clock (flows *f1*, *f2*, *f3*). In this case, the flow is lagging behind the global clock and the dequeued packet will not move the global clock forward according to Definition 5. The flow will have a chance to dequeue consecutive packets as long as the above condition holds, without an impact on the global clock.

(2) $v(p_f^{i-1}) \leq c(k) < v(p_f^i)$:
The dequeued packet has a virtual time later than the global clock, therefore, this packet can move the global clock to $v(p_f^i)$ (flows *f4*, *f5*). The flow can only send a single packet, since dequeuing any packet with a virtual time after the clock will force the scheduler to switch to a new sub-queue (see §3.1). The value of $v(p_f^i) - v(p_f^{i-1})$ is the size of packet p_f^{i-1}, and is therefore always smaller than the maximum packet size. Let M denote the maximum possible packet size in the scheduler, i.e., $M = max_{f \in Q}(l_f^{max})$. In this condition, the updated virtual clock has the following bound:

$$v(p_f^i) \leq v(p_f^{i-1}) + M \leq c(k) + M$$

$$c(k+1) = max_{f \in Q}(v(p_f^i)) \leq c(k) + M.$$

No other conditions are possible, Theorem 1 follows. □

Theorem 2 (**Maximum Burstiness of SCRR**)**.** At each scheduling round, the aggregate size of a set of packets dequeued from one sub-queue is at most $(r_{\max} + 1)M$, where $r_{\max} = \max_{f \in Q} r_f$.

Proof. We denote the burst size for flow f at round k as $Burst_f^k$. If the flow has become active during the scheduling round, or if the virtual clock of the dequeued packet is later than the global clock, it is only allowed to send a single packet, according to Theorem 1.

If the virtual clock of the dequeued packet is earlier than the global clock, the flow is lagging and all packets with virtual time earlier than the global clock can be dequeued. Given that the virtual time of these packets is later than the global clock of the previous round–otherwise they would have been dequeued during the previous scheduling round–, the set of packets dequeued is constrained by:

$$c(k-1) < v(p_f^i) \leq v(p_f^{i+j}) \leq c(k)$$

The virtual time difference between the first and last packet is bounded by the clock advance, which is bounded by the maximum packet size, M:

$$v(p_f^{i+j}) - v(p_f^i) \leq c(k) - c(k-1) \leq M \tag{3}$$

In a flow, the virtual clock increase of successive packets corresponds to the size of those packets. The virtual clock is based on packet start times (see Fig. 6b for a demonstration), so the clock increase is based on the previous packet. Since $v(p_f^i) - v(p_f^{i-1}) = l(p_f^{i-1})$ and by (3),

$$v(p_f^{i+j}) - v(p_f^i) = \sum_{n=i}^{i+j-1} l(p_f^n)$$

$$Burst_f^k = \sum_{n=i}^{i+j-1} r_{\max} \cdot l(p_f^n) + l(p_f^{i+j}) \leq (r_{\max}+1)M \quad \square$$

4.2 Throughput and Fairness in SCRR

In this section, we derive theoretical bounds for SCRR.

Theorem 3 (Fairness Index for SCRR). The fairness index of SCRR, FairnessIndex_f, is 1 for any flow, f.

Proof. Suppose each flow in our system is continuously backlogged and each packet is tagged with their start-time at enqueue. Let $A(k) = c(k+1) - c(k)$, $n = |Q|$, $M = \max_{f \in Q}\left(l_f^{\max}\right)$, and $R = \sum_{f \in Q} r_f$.

By Theorem 1, $A(k) \leq M$ for each round k. If a packet has a virtual time before or equal to $c(k)$, our algorithm will send it by the end of round k. Thus, the total amount of bytes sent by flow f by the end of round k, $\text{Sent}_{f,k}$, is bounded by

$$r_f \sum_{j=1}^{k} A(j) - M \leq r_f \sum_{j=1}^{k-1} A(j) \leq \text{Sent}_{f,k} \leq r_f \sum_{j=1}^{k} A(j) + M.$$

Similarly, summing over all flows provides bounds for Sent_k, the total amount of data sent overall by round k.

$$\sum_{f \in Q}\left(r_f \sum_{j=1}^{k} A(j) - M\right) \leq \text{Sent}_k \leq \sum_{f \in Q}\left(r_f \sum_{j=1}^{k} A(j) + M\right)$$

Then combining these inequalities gives us

$$\frac{r_f \sum_{j=1}^{k} A(j) - M}{R \sum_{j=1}^{k} A(j) + n \cdot M} \leq \frac{\text{Sent}_{f,k}}{\text{Sent}_k} \leq \frac{r_f \sum_{j=1}^{k} A(j) + M}{R \sum_{j=1}^{k} A(j) - n \cdot M}.$$

As the number of rounds approaches infinity, the fairness quotient converges to r_f/R by the squeeze theorem, and thus our fairness index is

$$\text{FairnessIndex}_f = \frac{FQ_f}{\frac{r_f}{\sum_{g \in Q} r_g}} = \frac{FQ_f}{\frac{r_f}{R}} = 1. \quad \square$$

5 Testbed Evaluations

We present a scalable way to evaluate SCRR by developing a novel approach to testing packet schedulers. We use a physical testbed with synthesized traffic and perform measurements at the application level. For simplicity, we only evaluate flow scheduling, i.e., all scheduling entities (flows) have the same weight. We believe that our findings apply to QoS scheduling, where flows have different weights. Our experiments yield the following findings:

- In the presence of NIC acceleration functions, SCRR reduces the CPU utilization of the packet scheduling operations by up to 46% and 23% compared to STFQ and DRR (with 1500B quantum), respectively (§5.3).
- SCRR achieves equivalent Jain fairness to all fair scheduling schemes when facing up to 20k active flows (§5.4). E.g., SCRR maintains a Jain index of > 0.97 when scheduling up to 20k active flows.
- SCRR offers 87×, 50%, and 18% lower latency for a request-response workload, compared to tail-drop, DRR-1500 with Sparse Flow Optimization, and STFQ, respectively (§5.5).
- SCRR offers 15× and 40% lower frame latency compared to *tail-drop* and *DRR+SFO-1500*, respectively, when scheduling synthetic VBR traffic (§5.5).

A more comprehensive set of results on performance of PIFO approximations (D.1), SCRR's individual components (D.2), application performance under TCP BBRv3 congestion control (D.3), CPU utilization and throughput (D.4), flow fairness (D.5) and application performance under TCP Cubic (D.6) can be found in Appendix D.

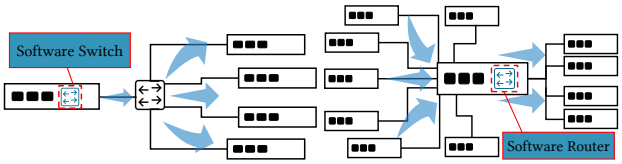
5.1 Scheduler Implementations

We implement various queuing disciplines and schedulers in Linux *tc* subsystem as a set of 'qdisc' modules for Linux version 5.10. These software implementations in the *tc* subsystem enable us to use the schedulers on a real system and employ them as part of a sender software switch, a software router, or combined with other qdiscs. Our implementations are compatible with all accelerations of the Linux TCP/IP stack, such as segmentation offloading and interrupt coalescing. Our implementations are intentionally single-threaded to remove the interference from the NIC driver's weighted round-robin scheduler among parallel packet rings [7, 84]. The evaluated systems are as follows:

Tail-drop uses the standard byte-FIFO from the Linux kernel (*bfifo*) on a single queue with the default capacity of 18 MB (15 ms of traffic) for a 10 Gbps setup. **PI2** is our own optimized version of the PI2 reference implementation [31]. As a representative of modern Active Queue Management (AQM) [73], PI2 uses a single queue and its target delay is 15 ms (the recommended default).

DRR+SFO is the Deficit Round Robin implementation from the *sch_fq* module [33]. We stripped all extraneous features from this scheduler and kept only the classifier and the scheduler. The simpler code and smaller sub-queue structure help dramatically reduce CPU usage. The classifier is based on a small hash table, and each hash bucket contains an RB-tree [10] holding the sub-queues. This implementation improves the original DRR algorithm [81] with *Sparse Flow Optimization* [47], which prioritizes newly active sub-queues to reduce the response times of latency-sensitive flows. We also include **DRR** in our results to represent the original DRR algorithm without Sparse Flow Optimization. This is the implementation widely found on hardware targets. The suffix of DRR+SFO and DRR indicates its quantum.

SCRR is our implementation of SCRR. This scheduler



(a) *Edge Switching* (b) *Software Routing*

Figure 7: *Topologies in the physical testbed.*

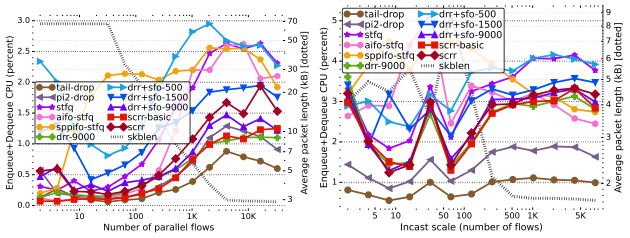
reuses the structure and classifier of the DRR scheduler, along with its sub-queue management and instrumentation. **SCRR** includes four enhancements: *No Packet Metadata* (§3.2), *Sparse Flow Optimization* (§3.3.1) *Initial Advance* (§3.3.2), and *No Empty* (§3.3.3). **SCRR-basic** refers to the SCRR algorithm without these enhancements.

STFQ is our implementation of STFQ [42]. We reuse the structure and classifier of the DRR and SCRR modules. The sub-queue pointers are stored in a sorted RB-tree [10], ordered by the virtual time of the head packets. Using an RB-tree for implementing STFQ proved to be the most efficient way to manage a dynamic sorted list– inspired by CFS [5].

AIFO and **SP-PIFO** are two approximations of Push-In First-Out queuing abstractions that use STFQ prioritization [17, 82, 89]. We implement AIFO using a single FIFO queue with default parameters set as 64 for the sampling and 10% for the burst headroom. SP-PIFO uses 8-level priority FIFOs with no restrictions on their size. We include both techniques in our results and refer the readers to Appendix D.1 for further analysis of these schemes,

5.2 Experiment Setup

We use the two topologies as depicted in Fig. 7. The edge switching topology (Fig. 7a) represents packet switching on a sender host machine, for example in a virtual switch deployed on a server in a datacenter or in the cloud. One server acts as the traffic generator and the schedulers are configured on that server to forward the traffic. This mimics a small-scale outcast traffic pattern setting. The software routing topology (Fig. 7b) is used by default and represents in-network scheduling where one software gateway routes traffic between the senders and receivers, this represents deployment on home routers, SD-WAN gateways, access points or middleboxes. The schedulers are configured on the middle server to forward the traffic between seven receiving network interfaces and one transmitting interface being the bottleneck. This topology enables us to emulate incast traffic patterns. These topologies are implemented in a testbed consisting of ten Linux servers connected via X710 10 Gbps NICs to Aruba 5406R switches. The scheduling server machine in the edge topology features 2x Intel Xeon E5-2643 CPUs totaling 24 logical cores (3.4 GHz). The scheduling server for the routing topology leverages 2x Intel Xeon E5-2680 CPUs totaling 28 cores (2.4 GHz). A subset of our experiments is done in a 25 Gbps topology consisting of four servers in



(a) *TSO* (b) *LRO*

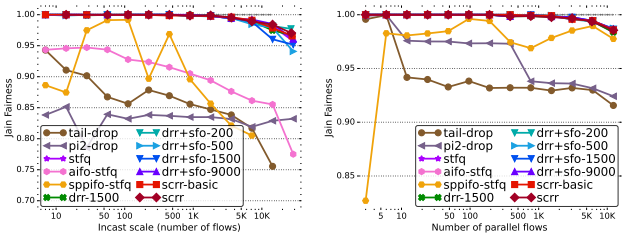
Figure 8: *Impact of increasing Incast scales on CPU utilization in the presence of NIC offloading functions.*

the edge-switching topology, connected via Intel E810 NICs and a Tofino 2 switch. By default, each experiment runs for 1 minute. The workload traffic is generated using versions 2.1.9 and 2.1.10 of *iperf* [8]. The default TCP congestion control is Cubic (refer to §D.3 for BBRv3 results). The default MSS is 1456 bytes and LRO/GRO are disabled unless specified otherwise. In some experiments, Byte Queue Limits are lowered [48], to shrink the NIC transmit queue and to reduce transmit batching, which can decrease application latency. Finally, CPU utilization is measured using Linux *perf* framework [9] by summing the usage of the enqueue and dequeue functions of each scheduler.

5.3 NIC Accelerations - TSO & LRO

We start by investigating the interaction between hardware offloading functions and the schedulers. Fig. 8a shows the impact of TCP Segmentation Offloading (TSO) on software packet schedulers as we increase the number of TCP flows in the edge switching topology (Fig. 7a). With a larger number of flows, the individual bandwidth share decreases and the average TSO packet size (dotted line) decreases from 64 kB to 1500B (full packet size distribution is in Fig. 4a), causing an increase in CPU usage in all schedulers. Decreasing the quantum of *DRR* increases CPU cycles wasted in missed visits (§2.4). The CPU utilization of *SP-PIFO* and *AIFO* is atypical and described in §D.1.1. We also observe that *SCRR-basic* scheduling automatically lowers CPU usage by adapting to the packet sizes, yielding a similar CPU overhead as *DRR-9000*. This experiment is a worst case for *SCRR*, in our other experiments it often has equal or lower CPU usage than *DRR+SFO-9000* (§5.5). Nevertheless, with 2048 flows, *SCRR* reduces the CPU utilization compared to *STFQ* and *DRR-1500* by 46% and 23%, respectively.

Fig. 8b presents the LRO packet sizes (dotted) received on the router in topology 7b, where each of the two sender machines produces half of the workload flows. The impact of LRO is a lot less predictable than TSO, and less packet aggregation is done (full packet size distribution is in Fig. 4b). *DRR* with smaller quanta cannot adapt to the dynamics of the traffic, causing CPU wastage while *SCRR* offers lower CPU overhead amongst schedulers with flow classification at high scales. Our previous results in Fig. 5b indicate that for



(a) *TCP flows and MSS bias* (b) *UDP flows and rate bias*

Figure 9: *Jain fairness index with different bias settings.*

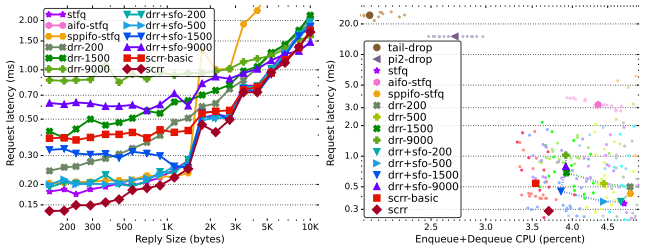
every 1% increase in the overall CPU utilization of the qdisc up to 17% reduction in throughput can be expected. In other experiments, we disable the offloading functions to prevent the schedulers from dealing with unwanted large segments.

5.4 Application Fairness

The crucial property of a packet scheduler is enforcing fairness and QoS. To better scale the experiments, we configured all sub-queues with identical weights and configured biases between flows to showcase how the scheduler overcome the bias and achieves fairness under backlogged flows.

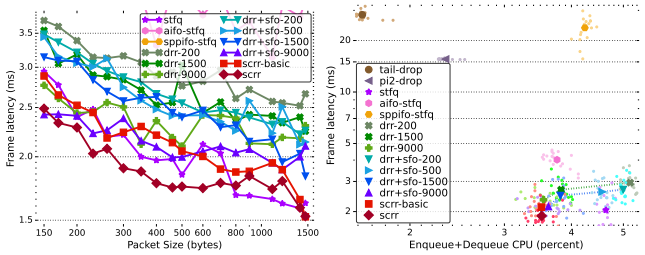
Fig. 9a uses maximum segment sizes (MSS) as the bias between TCP flows, each of the seven senders is using an MSS set to 2500, 3000, 3500, 5000, 6000, 8500, and 8956 bytes, respectively. This forces the schedulers to send more packets from flows that feature smaller packets and can verify the proper byte-level accounting of the schedulers. On each sender, the number of TCP connections varies from 1 to 4096, for a maximum of 28,672 flows. The Jain Index [49] based on the throughput of every flow shows that all Fair Queuing and Round-Robin schedulers, including *SCRR*, offer superior fairness as they enforce byte-level fairness for every flow. Since TCP congestion control is not designed to provide byte-level fairness, *tail-drop* and *PI2* show poor fairness. *AIFO* and *SP-PIFO* also show poor fairness due to their limited number of sub-queues. At a very high number of TCP flows (28k), the fairness of all schedulers is impacted by hash collisions in the classifier, a well-known caveat of using hashing in flow classifiers. Most QoS classifiers have well-defined QoS classes that do not use hashing and would not suffer from this issue. With 28k flows, some flows experience reduced performance or stalls, especially with the *tail-drop* queue.

We repeat the above experiments using UDP and flow rates as the bias in a 25 Gbps setup consisting of one sending machine and three receivers. Each connection pair is configured to offer 16.6, 12.5, and 8.3 Gbps, respectively, all bottlenecked by the sender's 25 Gbps egress bandwidth. According to Fig. 9b, a similar trend can be observed for all evaluated schedulers. The only difference is *AIFO*, which is able to throttle faster flows in order to control the queue utilization. A more comprehensive set of results on fairness using UDP and different bias settings can be found in Appendix D.5.



(a) *Response size vs latency* (b) *CPU util. vs latency*

Figure 10: *Impact of packet scheduling on request latency.*



(a) *Frame size vs latency* (b) *CPU util. vs latency*

Figure 11: *Impact of packet scheduling on VBR frame latency.*

5.5 Bursty Flow Performance

Request-response workloads. Modern networks are filled with small request-response flows [22, 62, 71]. This improved BufferBloat experiment [38] showcases how our schedulers can help those small requests when they are mixed with long-lived heavy flows. Here, two of the receiver machines generate 8 latency-sensitive flows each, the flows are a sequences of short back-to-back request-response with a request of 50B and varying response lengths. The replies are usually a short burst: the TCP ack and one or more packets for the reply. Another two senders generate 32 parallel background long-lived TCP flows each, to force the scheduler to be congested (uncongested schedulers just act as FIFOs).

Fig. 10a presents the latency of a selected subset of the schedulers. As the response size increases, they are split across more TCP segments, requiring more scheduling rounds to be forwarded, and we see a corresponding increase in latency. *SP-PIFO* has poor latency and fairness beyond one MSS, this is detailed in §D.1.3. *DRR+SFO* variants with smaller quanta are able to produce lower response times as smaller response packets experience lower queuing delays, similar to *STFQ*. *SCRR* with its latency enhancements offers superior performance compared to all other alternatives because when a sub-queue is added to the schedule, it can send multiple packets in the round instead of a single packet.

Fig. 10b averages CPU overhead and average latency over all reply sizes of fig. 10a. *Tail-drop* and *PI2* use a single queue that is quickly filled by background TCP flows, imposing a full queuing delay to the request packets. *AIFO* keeps its single queue short (§D.1.2), which only slightly helps its latency at the cost of increased CPU utilization. *SP-PIFO*'s CPU utilization is amongst the highest, and higher than *STFQ*

(§D.1.1). The performance trade-offs of *DRR* is detailed in Fig. 5a. In this experiment, 500B looks to be a sweet spot for quanta setting. Other configurations may lead to different sweet spots (see §D.6). Both *SCRR* and *SCRR-basic* adapt to packet sizes and are therefore able to offer smaller latency than the optimally configured *DRR+SFO-1500*. Specifically, *SCRR* is able to achieve lower CPU utilization than all *DRR* counterparts while improving the application latency by 87×, 1.5×, and 1.18×, compared to *tail-drop*, *DRR+SFO-1500*, and *STFQ*, respectively. A more comprehensive performance study of SCRR components can be found in Appendix D.2.

Streaming workloads. Streaming workloads are nowadays popular [55]. They commonly use *Variable Bit Rate (VBR)* encoding and reduce their packet sizes to mitigate the impact of packet losses [30, 79]. Those short bursts of small packets will alternate between sparse and backlogged, demanding low latency and high throughput performance from the scheduler. We use the *isochronous* capability in *Iperf* to emulate 64 concurrent VBR UDP flows on two servers, each targeting 1.5 Mbps and 30 Frames Per Second (FPS), which represents a typical 720p Zoom video communication bitrate. The VBR flows are competing in 7b topology with 128 long-running heavy TCP flows from two servers. Fig. 11 presents the one-way latency of each frame for different schedulers. Reducing packet size decreases performance as the header and processing overheads of small packets hurts their throughput. For *DRR*, the increased CPU utilization of short quantum hurts the throughput of the burst. *STFQ* manages low latency despite its high CPU utilization. Much like the request-response workload, the majority of VBR frames are bursty, therefore, *SCRR* which is carefully designed for bursty flow performance offers superior latency compared to all alternatives. Concretely, *SCRR* offers 15×, 1.4×, and 1.08× lower frame latency compared to *tail-drop*, *DRR+SFO-1500*, and *STFQ*, respectively. Our experiments with BBRv3 congestion control in Appendix D.3 further confirm that while congestion control has no impact on the performance of short bursty flows, *SCRR* can provide lower latency and resource utilization compared to the state-of-the-art schedulers.

6 Related Work

Starting from Nagle's [64] practical fair queuing, packet scheduling has been the focus of numerous works.

The Origin of Self-Clocking. Packet-based Generalized Processor Sharing (*PGPS*) scheduling [68] studies latency and burstiness bounds with virtual clocking and leaky bucket admission control. Self-Clocked Fair Queuing (*SCFQ*) [41] tags packets with a virtual clock, indicating their *virtual finish time* (expected end of transmission). To determine eligible queues for transmission, the scheduler maintains a sorted list based on virtual times of head packets. Upon enqueue, each packet's virtual time is calculated as the queue's current virtual time, augmented by the packet's length, proportional to the queue's assigned weight assigned. After transmission, the scheduler updates its virtual time to the transmitted packet's time.

Start-Time Fair Queuing (*STFQ*) [42] is a similar fair queuing paradigm, using virtual clocking but tags packets with their virtual start time (expected start of transmission). This virtual time equals the finish time of the tail packet if the queue is backlogged or the scheduler's global virtual time if idle. *STFQ* improves *SCFQ*'s average scheduling delays by 53% [42], but its O($log(n)$) computational complexity limits its practicality. Several works explore *STFQ*'s implementation [45, 53, 88] or attempt to approximate it in programmable hardware [17, 63, 89]. However, scalability challenges remain [45, 82], and existing approximations fail to ensure flow fairness and ideal sparse flow performance. Fair queuing schedulers have strong performance, and may outperform SCRR for some workloads or configurations. Finally, *DRR* variants [56, 69, 81] offer flow-level fairness, but using a user-specified quantum makes it difficult to argue their performance trade-offs.

Programmable and priority packet scheduling. [78] implements a programmable queuing abstraction for complex scheduling and queue management paradigms such as *Weighted Fair Queuing*. Similarly, *PIFO* [82] enables packet-level prioritization at a limited scale in hardware. As shown in Appendix D.1, existing programmable scheduling implementations suffer from under-utilization and heavy packet reordering. *AFQ* [77] similarly approximates fair queuing paradigms in programmable hardware. *Karuna* [28] presents packet scheduling for deadline flows. *SCRR* focuses on standard TCP/IP networks while allowing for future extensions.

7 Conclusions

Dating back to almost three decades ago, deficit round-robin is still one of today's widely used fair queueing paradigms. This paper identifies the diversity in packet sizes and burstiness as two critical characteristics of modern Internet traffic that make DRR's performance unpredictable and sub-optimal. We then revisit classical fair queueing and introduce the *Self-Clocked Round Robin Scheduler (SCRR)* with strong fairness bounds. We augment SCRR with enhancements that target bursty, latency-sensitive flows over the Internet. Our physical testbed evaluations, carefully crafted to probe the performance of the packet scheduler, reveal that *SCRR* offers lower resource consumption and superior latency performance compared to widely deployed scheduling and policing mechanisms.

Acknowledgments

We thank our shepherd John Sonchack and all the anonymous reviewers for their constructive feedback. We thank Hewlett Packard Labs for hosting the first two authors as interns during this work. The work is partially supported by NSF CNS NeTS grant 2313164.

References

[1] U.S. National CyberWatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) Traces. https://www.netresec.com/?page=MACCDC, 2012.

[2] SQM, Cake and Piece of Cake QoS - High CPU Usage. https://forum.openwrt.org/t/sqm-cake-and-piece-of-cake-qos-high-cpu-usage/17794, 2018.

[3] SQM/QoS Can Saturate the CPU/Is This Expected or Can the Code Be Improved? https://forum.openwrt.org/t/sqm-qos-can-saturate-the-cpu-is-this-expected-or-can-the-code-be-improved/, 2019.

[4] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset, 2019.

[5] CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html, 2023.

[6] Intel® Ethernet Controller E810 Datasheet. https://cdrdv2.intel.com/v1/dl/getContent/613875?wapkw=columbiaville%20datasheet, 2023.

[7] Intel® Ethernet Controller X710/ XXV710/XL710 Datasheet. https://cdrdv2-public.intel.com/332464/332464_710_Series_Datasheet_v_4_1.pdf, 2023.

[8] Iperf 2 Download. https://sourceforge.net/projects/iperf2/, 2023.

[9] perf: Linux Profiling With Performance Counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2023.

[10] Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt, 2023.

[11] Segmentation Offloads in Linux Kernel. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html, 2023.

[12] Understand and Configure MDRR/WRED on 12000 Series Routers. https://www.cisco.com/c/en/us/support/docs/routers/12000-series-routers/18841-mdrr-wred-18841.html, 2023.

[13] BBR. https://github.com/google/bbr, 2024.

[14] Configuring MDRR on Enhanced Queuing DPCs. https://www.juniper.net/documentation/us/en/software/junos/cos/topics/concept/cos-configuring-mdrr-on-enhanced-queuing-dpcs.html, 2024.

[15] Linux Advanced Routing & Traffic Control HOWTO. https://tldp.org/HOWTO/Adv-Routing-HOWTO, 2024.

[16] AGARWAL, S., CAI, Q., AGARWAL, R., SHMOYS, D., AND VAHDAT, A. Harmony: A Congestion-free Datacenter Architecture. In *NSDI* (2024).

[17] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. SP-PIFO: Approximating Push-in First-out Behaviors Using Strict-priority Queues. In *NSDI* (2020).

[18] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Mminimal Near-optimal Datacenter Transport. In *SIGCOMM* (2013).

[19] ARASHLOO, M. T., BECKETT, R., AND AGARWAL, R. Formal Methods for Network Performance Analysis. In *NSDI* (2023).

[20] BAHRAMALI, A., SOLTANI, R., HOUMANSADR, A., GOECKEL, D., AND TOWSLEY, D. Practical Traffic Analysis Attacks on Secure Messaging Applications. In *NDSS* (2020).

[21] BENNETT, J. C. R., AND ZHANG, H. WF2Q: Worst-case Fair Weighted Fair Queueing. In *INFOCOM* (1996).

[22] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[23] BERG, B., BERGER, D. S., MCALLISTER, S., GROSOF, I., GUNASEKAR, S., LU, J., UHLAR, M., CARRIG, J., BECKMANN, N., HARCHOL-BALTER, M., AND GANGER, G. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI* (2020).

[24] BOURON, J., CHEVALLEY, S., LEPERS, B., ZWAENEPOEL, W., GOUICEM, R., LAWALL, J., MULLER, G., AND SOPENA, J. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *ATC* (2018).

[25] CAI, Q., CHAUDHARY, S., VUPPALAPATI, M., HWANG, J., AND AGARWAL, R. Understanding Host Network Stack Overheads. In *SIGCOMM* (2021).

[26] CAI, Q., VUPPALAPATI, M., HWANG, J., KOZYRAKIS, C., AND AGARWAL, R. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM* (2022).

[27] CHANG, H., VARVELLO, M., HAO, F., AND MUKHERJEE, S. Can You See Me Now? A Measurement Study of Zoom, Webex, and Meet. In *IMC* (2021).

[28] CHEN, L., CHEN, K., BAI, W., AND ALIZADEH, M. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *SIGCOMM* (2016).

[29] CHO, K., MITSUYA, K., AND KATO, A. Traffic Data Repository at the WIDE Project, 2000.

[30] CHOI, A., KARAMOLLAHI, M., WILLIAMSON, C., AND ARLITT, M. Zoom Session Quality: A Network-Level View. In *PAM* (2022).

[31] DE SCHEPPER, K., ALBISSER, O., STEEN, H., AND TILMANS, O. sch_dualpi2: Dual Queue with Proportional Integral controller Improved with a Square. https://github.com/L4STeam, 2020.

[32] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM* (1989).

[33] DUMAZET, E. sch_fq: Fair Queue Packet Scheduler. https://lwn.net/Articles/564825/, 2013.

[34] DUMAZET, E., AND LI, C. BIG TCP. In *Netdev 0x15* (2021).

[35] FELDMANN, A., GASSER, O., LICHTBLAU, F., PUJOL, E., POESE, I., DIETZEL, C., WAGNER, D., WICHTLHUBER, M., TAPIADOR, J., VALLINA-RODRIGUEZ, N., HOHLFELD, O., AND SMARAGDAKIS, G. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *IMC* (2020).

[36] FLOYD, S., HENDERSON, T., AND GURTOV, A. The NewReno Modification to TCP's Fast Recovery Algorithm. https://datatracker.ietf.org/doc/html/rfc3782, 2001.

[37] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An Open-Source 100-Gbps NIC. In *FCCM* (2020).

[38] GETTYS, J. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing* (2011).

[39] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A Microscopic View of Bursts, Buffer Contention, and Loss in Data Centers. In *IMC* (2022).

[40] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource Fair Queueing for Packet Processing. In *SIGCOMM* (2012).

[41] GOLESTANI, S. J. A Self-clocked Fair Queueing Scheme for Broadband Applications. In *INFOCOM* (1994).

[42] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time Fair Queueing: a Scheduling Algorithm for Integrated Services Packet Switching Networks. *ToN* (1997).

[43] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. SmartNIC Performance Isolation with FairNIC. In *SIGCOMM* (2020).

[44] HE, J., AMMAR, M., ZEGURA, E., AND HALEPOVIC, E. QoE Metrics for Interactivity in Video Conferencing Applications: Definition and Evaluation Methodology. In *MMSys* (2024).

[45] HEDAYATI, M., SHEN, K., SCOTT, M. L., AND MARTY, M. Multi-Queue Fair Queuing. In *ATC* (2019).

[46] HEMMINGER, S. tc-netem(8): Network Emulator - Linux man Page. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[47] HØILAND-JØRGENSEN, T. Analyzing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm. *IEEE Communications Letters* (2018).

[48] HRUBY, T. Byte Queue Limits Revisited. In *Linux Plumbers Conference* (2012).

[49] JAIN, R., CHIU, D. M., AND WR, H. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR* (1998).

[50] JIANG, H., AND DOVROLIS, C. Why Is the Internet Traffic Bursty in Short Time Scales? In *SIGMETRICS* (2005).

[51] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT* (2013).

[52] KARAMOLLAHI, M., WILLIAMSON, C., AND ARLITT, M. Packet-Level Analysis of Zoom Performance Anomalies. In *ICPE* (2023).

[53] KORTEBI, A., MUSCARIELLO, L., OUESLATI, S., AND ROBERTS, J. Minimizing the Overhead in Implementing Flow-aware Networking. In *INCS* (2005).

[54] KUZNETSOV, A. N. tc-tbf(8): Token Bucket Filter - Linux man Page. https://man7.org/linux/man-pages/man8/tc-tbf.8.html.

[55] LEE, I., LEE, J., LEE, K., GRUNWALD, D., AND HA, S. Demystifying Commercial Video Conferencing Applications. In *MM* (2021).

[56] LENZINI, L., MINGOZZI, E., AND STEA, G. Aliquem: a Novel DRR Implementation to Achieve Better Latency and Fairness at O(1) Complexity. In *IEEE QoS* (2002).

[57] LEVY, M. Jumbo Frame Deployment at Internet Exchange Points. https://www.ietf.org/archive/id/draft-mlevy-ixp-jumboframes-00.txt, 2011.

[58] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving Resource Efficiency at Scale. In *ISCA* (2015).

[59] MACMILLAN, K., MANGLA, T., SAXON, J., AND FEAMSTER, N. Measuring the Performance and Network Utilization of Popular Video Conferencing Applications. In *IMC* (2021).

[60] MENG, Z., ATRE, N., XU, M., SHERRY, J., AND APOSTOLAKI, M. Confucius: Achieving Consistent Low Latency With Practical Queue Management for Real-time Communications. *arXiv cs.NI 2310.18030* (2023).

[61] MICHEL, O., SENGUPTA, S., KIM, H., NETRAVALI, R., AND REXFORD, J. Enabling Passive Measurement of Zoom Performance in Production Networks. In *IMC* (2022).

[62] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *SIGCOMM* (2018).

[63] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM* (2016).

[64] NAGLE, J. On Packet Switches with Infinite Storage. *IEEE ToC* (1987).

[65] NISTICÒ, A., MARKUDOVA, D., TREVISAN, M., MEO, M., AND CAROFIGLIO, G. A Comparative Study of RTC Applications. In *IS* (2020).

[66] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *SOSP* (2013).

[67] PANG, R., ALLMAN, M., BENNETT, M., LEE, J. R., PAXSON, V., AND TIERNEY, B. A First Look at Modern Enterprise Traffic. In *IMC* (2005).

[68] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *ToN* (1993).

[69] RAMABHADRAN, S., AND PASQUALE, J. Stratified round Robin: a low complexity packet scheduler with bandwidth fairness and bounded delay. In *SIGCOMM* (2003).

[70] RAMAKRISHNAN, K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. https://datatracker.ietf.org/doc/html/rfc3168, 2001.

[71] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *SIGCOMM* (2015).

[72] SAEED, A., DUKKIPATI, N., VALANCIUS, V., THE LAM, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[73] SCHEPPER, K., ALBISSER, O., TSANG, I., AND BRISCOE, B. PI2 : A Linearized AQM for both Classic and Scalable TCP. In *CoNEXT* (2016).

[74] SCHLINKER, B., CUNHA, I., CHIU, Y.-C., SUNDARESAN, S., AND KATZ-BASSETT, E. Internet Performance from Facebook's Edge. In *IMC* (2019).

[75] SENGUPTA, S., YADAV, V. K., SARAF, Y., GUPTA, H., GANGULY, N., CHAKRABORTY, S., AND DE, P. MoViDiff: Enabling Service Differentiation for Mobile Video Apps. In *IM* (2017).

[76] SHARAFZADEH, E., ABDOUS, S., AND GHORBANI, S. Understanding the Impact of Host Networking Elements on Traffic Bursts. In *NSDI* (2023).

[77] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI* (2018).

[78] SHARMA, N. K., ZHAO, C., LIU, M., KANNAN, P. G., KIM, C., KRISHNAMURTHY, A., AND SIVARAMAN, A. Programmable Calendar Queues for High-Speed Packet Scheduling. In *NSDI* (2020).

[79] SHARMA, T., MANGLA, T., GUPTA, A., JIANG, J., AND FEAMSTER, N. Estimating WebRTC Video QoE Metrics Without Using Application Headers. In *IMC* (2023).

[80] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the Data Center Network. In *NSDI* (2011).

[81] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[82] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *SIGCOMM* (2016).

[83] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI* (2019).

[84] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *ATC* (2017).

[85] THAKKAR, P., SAXENA, R., AND PADMANABHAN, V. N. AutoSens: Inferring Latency Sensitivity of User Activity Through Natural Experiments. In *IMC* (2021).

[86] WANG, Z., LI, Z., LIU, G., CHEN, Y., WU, Q., AND CHENG, G. Examination of WAN traffic characteristics in a large-scale data center network. In *IMC* (2021).

[87] XU, X., AND CLAYPOOL, M. Measurement of Cloud-based Game Streaming System Response to Competing TCP Cubic or TCP BBR Flows. In *IMC* (2022).

[88] XU, Y., AND ZHAO, M. IBIS: Interposed Big-data I/O Scheduler. In *HPDC* (2016).

[89] YU, Z., HU, C., WU, J., SUN, X., BRAVERMAN, V., CHOWDHURY, M., LIU, Z., AND JIN, X. Programmable packet scheduling with a single queue. In *SIGCOMM* (2021).

[90] YUAN, X., AND DUAN, Z. Fair Round-Robin: A Low Complexity Packet Scheduler with Proportional and Worst-Case Fairness. *IEEE ToC* (2009).

[91] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEIJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., PENNA, P. H., DEMOULIN, M., CHOUDHURY, P., AND BADAM, A. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP* (2021).

[92] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-resolution Measurement of Data Center Microbursts. In *IMC* (2017).

[93] ZHANG, Y., TAN, Y., STEPHENS, B., AND CHOWDHURY, M. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI* (2022).

A Self-Clocked Round-Robin Scheduling

This section expands on the design of the Self-Clocked Round-Robin (SCRR) packet scheduler. Both *SCRR-basic* and *SCRR* with its enhancements are presented.

Algorithm 1 SCRR-basic Enqueue

```
function ENQUEUE(pkt)
    cur_queue ← classify(pkt)
    if cur_queue is inactive then
        sub_queues.tail ← cur_queue
        rounds ← rounds + 1
    end if
    pkt.virtual ← max(virtual_clock, cur_queue.clock)
    cur_queue.clock ← pkt.virt_tag + pkt.length
    cur_queue.enqueue(pkt)
end function
```

Algorithm 2 SCRR-basic Dequeue

```
Require: NP ≥ 0                                   ▷ Packet Count
function DEQUEUE                                  ▷ Packet Retrieval
    if NP == 0 then return NULL
    end if
    cur_queue ← sub_queues.head
    pkt ← cur_queue.dequeue()
    NP ← NP − 1
                                                  ▷ Global Clock Advance
    if NP == 0 then
        virtual_dequeue ← pkt.virtual + pkt.length
        virtual_clock ← virtual_dequeue
    else if pkt.virtual > virtual_dequeue then
        virtual_dequeue ← pkt.virtual
    end if
                                                  ▷ Sub-Queue Round Robin
    virtual_next ← pkt.virtual_time + pkt.length
    if virtual_next > virtual_clock or cur_queue is empty
then
        if rounds ≤ 0 then
            virtual_clock ← virtual_dequeue
            rounds ← all_queues.length
        end if
        rounds ← rounds − 1
        sub_queues.head ← cur_queue.next
        if cur_queue is not empty then
            sub_queues.tail ← cur_queue
        end if
    end if
     return pkt
end function
```

A.1 Pseudo code of SCRR-basic

SCRR-basic is the simplest of all the variants of SCRR. The pseudo-code for the **enqueue process** of SCRR-basic is shown in Algorithm 1. This is the process that is run when a new packet arrives at the scheduler. First, the arrived packet passes through the classifier to obtain its sub-queue handle, i.e. the sub-queue where it will be enqueued. When doing per-flow classification, the 5-tuple from the packet header is used to select a sub-queue (source IP, destination IP, protocol, source transport port and destination transport port). Other QoS classifiers can also be used with SCRR. The current sub-queue may be active (already in the schedule) or inactive (empty). In the case the sub-queue is inactive, it is added at the end of the schedule, the schedule is a simple linked list of sub-queues. Tagging a packet with its virtual time is implementation dependent and usually done in the metadata associated with the packet. The packet virtual time is computed here exactly like *STFQ* [42], i.e., the most recent of the global clock and the finish time of the previous packet added to the sub-queue. The virtual time of the sub-queue is updated with the finish time of the new packet by adding its length in bytes. In the weighted version of SCRR, the packet length would be multiplied by the sub-queue weight. Ultimately, the packet is added to the SKB list of the corresponding sub-queue.

The pseudo-code for the **dequeue process** of SCRR-basic is shown in Algorithm 2. This is the process that is run when a packet needs to be sent over the outgoing link. First, a packet is extracted from the current sub-queue. Then, the global clock advance is computed. While conceptually similar to *STFQ* [42], SCRR makes sure that the clock can only go forward. Then, to advance the global clock, the scheduler maintains a temporary *virtual_dequeue* clock that tracks the largest virtual time of packets dequeued during the scheduling round. After finishing a scheduling round, the global clock is set to *virtual_dequeue*. The next part is where SCRR distinguishes itself from *STFQ*. SCRR uses the virtual time of the next packet in the sub-queue to decide if it stays on that sub-queue or round robin to the next sub-queue. If the virtual time is in the future (greater than current clock), or if the sub-queue is empty, then the scheduler moves on to the next sub-queue (that will be used next time the function is called). If a number of scheduling rounds equal to the number of sub-queues has elapsed, the virtual clock is updated with *virtual_dequeue* (the largest of the packet virtual times dequeued during that scheduling round). If the sub-queue is not empty, it is put at the back of the schedule, so that its packets can be sent at the next scheduling round.

The dequeue process must properly keep track of scheduling rounds. The enqueue process may add a previously inactive sub-queue to the current scheduling round. Tracking the scheduling round is needed to ensure that the global clock is only updated between rounds. If the global clock were to be

Algorithm 3 SCRR Enqueue (with enhancements)

```
function ENQUEUE(pkt)
    cur_queue ← classify(pkt)
    if cur_queue is inactive then
                                                ▷ No Empty
        if cur_queue.clock <= virtual_clock or nq == 0
then                                  ▷ Sparse Flow Optimization
            new_queues.tail ← cur_queue
            rounds ← rounds + 1
        else
            old_queues.tail ← cur_queue
        end if
    end if
    cur_queue.enqueue(pkt)
end function
```

updated after each packet transmission, the later sub-queues might fall further behind the global clock, leading to large unwanted packet bursts.

The algorithm describes SCRR-basic using the *start-time* of the packet like STFQ [42]. SCRR can also be implemented using the *finish-time* of the packet, similar to SCFQ [41]. We chose the *start-time* because it enables us to make SCRR's implementation simpler by computing the virtual time of the next packet without any information about that packet.

A.2 Pseudo code of SCRR with enhancements

SCRR, for brevity, refers to the variant of SCRR scheduler that includes all the enhancements, *No Packet Metadata*, *Sparse Flow Optimization*, *Initial Advance* and *No Empty*. In this section, we will only highlight the differences from SCRR-basic in §A.1.

The pseudo-code for the **enqueue process** of SCRR is shown in Algorithm 3. SCRR uses the *No Packet Metadata* enhancement, therefore the packet virtual time is not computed in the enqueue process, but in the dequeue process. Consequently, the packet is not tagged with its virtual time. In case the sub-queue has been inactive, SCRR uses the *No Empty* enhancement to decide how to schedule that sub-queue. If the finish time of the last packet on this sub-queue is in the past (earlier than the current clock), then the new packet can be fit into the schedule. In this case, it is eligible to use *Sparse Flow Optimization* and the inactive sub-queue is added to the list of sub-queue for priority scheduling. Otherwise, it is added at the end of the schedule.

The pseudo-code for the **dequeue process** of SCRR-basic is shown in Algorithm 4. SCRR uses *Sparse Flow Optimization*, therefore it first looks for a sub-queue in the list of previously inactive sub-queues, and only if that list is empty, it looks in the regular schedule. With *No Packet Metadata*, packet virtual time can be computed in dequeue. If the virtual finish of the last packet on the sub-queue is older than the virtual clock of the previous scheduling round, then that sub-queue was inactive. If the queue was inactive, and *Initial*

Algorithm 4 SCRR Dequeue (with enhancements)

```
Require: NP ≥ 0                                  ▷ Packet Count
Require: nq ≥ 0                    ▷ Active Sub-queue Count
function DEQUEUE                                 ▷ Packet Retrieval
    if NP == 0 then return NULL
    end if
                                      ▷ Sparse Flow Optimization
    cur_queue ← new_queues.head()
    if cur_queue is null then
        cur_queue ← old_queues.head()
    end if
    pkt ← cur_queue.dequeue()
    NP ← NP − 1
                                                ▷ No Packet Metadata
    if cur_queue.clock < virtual_previous then
        if config.initial_advance then          ▷ Initial Advance
            virtual_pkt ← virtual_previous + pkt.length
        else
            virtual_pkt ← virtual_clock
        end if
    else
        virtual_pkt ← cur_queue.clock
    end if
    virtual_next ← virtual_pkt + pkt.length
    cur_queue.clock ← virtual_next
                                          ▷ Global Clock Advance
    if NP == 0 then
        virtual_dequeue ← pkt.virtual + pkt.length
        virtual_previous ← virtual_clock
        virtual_clock ← virtual_dequeue
    else if pkt.virtual > virtual_dequeue then
        virtual_dequeue ← pkt.virtual
    end if
                                        ▷ Sub-queue Round Robin
    if virtual_next > virtual_clock or cur_queue is empty
then
        if rounds ≤ 0 then
            virtual_previous ← virtual_clock
            virtual_clock ← virtual_dequeue
            rounds ← nq
        end if
        rounds ← rounds − 1
        rr_queue ← rr_queue.next
        if cur_queue is not empty then
            old_queues.tail ← cur_queue
        end if
    end if
     return pkt
end function
```

Advance is enabled, the packet virtual time is based on the virtual clock of the previous scheduling round, this enables the sub-queue to potentially send a few packets in the current scheduling round before its clock advances beyond the global clock. Otherwise, it uses the current virtual clock, and the sub-queue can only send one packet in the current schedule. If the sub-queue has been active, the packet virtual time is based on the virtual time of the previous packet in that sub-queue, like in *SCRR-basic* and *STFQ*. After that, the sub-queue's clock is advanced based on the packet length and optionally the sub-queue's weight. The rest of the dequeuing process is identical to *SCRR-basic*, except the need to keep track of the global clock of the previous scheduling round.

B Extended Theoretical Analysis

This section expands the theoretical properties of the scheduler, extending our findings in §4. We present the analysis of SCRR using finish-time semantics.

B.1 Analysis of Backlogged SCRR Using Finish-time Semantics

Section 4.1 studies the SCRR scheduler using start time, similar to STFQ [42]. SCRR can also use finish time, similar to SCFQ [41]. Using the finish time in SCRR does not change the way the schedule behaves and similar proofs can be obtained for SCRR's fairness.

In Theorem 1, the packet virtual time is $v(p_f^i)$ and is equal to its finish-time (Fig. 12). In case (2), The value of $v(p_f^i) - v(p_f^{i-1})$ is the size of packet p_f^i (instead of packet p_f^{i-1}), this packet is still smaller than the maximum packet size, therefore the bound for case (2) is:
$v(p_f^i) \leq v(p_f^{i-1}) + M \leq c(k) + M$.
$c(k+1) = max_{f \in Q}(v(p_f^i)) \leq c(k) + M$.

The bound for case (2) is unchanged, and the other case is unchanged, therefore Theorem 1 also applies when using finish time.

In Theorem 2, equation 3 remains unchanged. Since the virtual clock is based on packet's finish time (Fig. 13), the increase in the global clock occurs based on the current packet (instead of the previous packet) :

$$v(p_f^{i+j}) - v(p_f^i) = \sum_{n=i+1}^{i+j} l(p_f^n)$$

$$Burst_f^k = \sum_{n=i}^{i+j} l(p_f^n) = l(p_f^i) + \sum_{n=i+1}^{i+j} l(p_f^n)$$

$$Burst_f^k \leq M + M$$

Throughput Bounds and Fairness. The proof of fairness with the finish time is identical to the proof using start-time (§4.2). The only difference is that we define $A(k) = c(k) - c(k-1)$ (instead of $A(k) = c(k+1) - c(k)$). The rest of the proof is unchanged, and as the number of rounds approaches infinity, the fairness quotient converges to $\frac{1}{n}$ and the fairness index converges to 1.

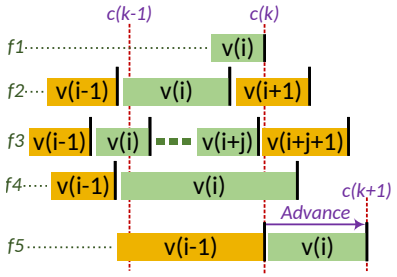


Figure 12: *Advancing the global clock in SCRR with finish-time semantics.*

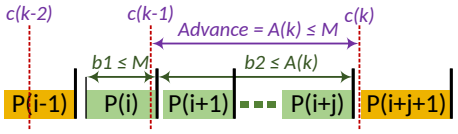


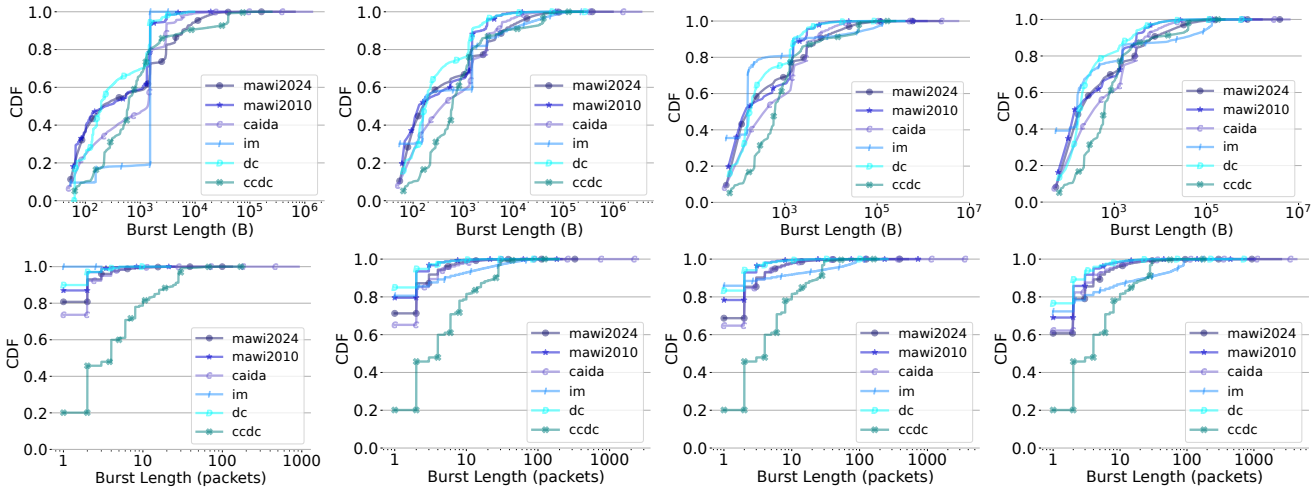
Figure 13: *Burstiness of SCRR with finish-time semantics.*

C Burstiness in Internet Traces

A burst is a train of packets with inter-packet gaps smaller than a threshold [51]. Using this definition, we plot the CDF of burst lengths for six representative Internet traces [1, 4, 20, 22, 29] in Fig. 14. Regardless of the threshold, a diverse range of burst lengths can be observed which suggests that Internet workloads feature both heavily backlogged and short flows arriving at the bottlenecks. In fact. the similarity among burst size distributions with different Inter-Packet Gap (IPG) thresholds suggest that our analysis of Internet bursts in §2 is independent of the chosen threshold. Additionally, the above figures highlight the presence of back-to-back packets with very small gaps, forming up a burst, and large inactivity gaps between bursts. This can be partly attributed to factors such as user behavior, especially for workloads like video streaming. We explore the performance of packet scheduling under VBR workloads in §5.5 and more thoroughly in Appendices D.3 and D.6.

D Extended Testbed Evaluation

This section presents an extended version of the testbed evaluation in §5, putting together our results of scheduling in the presence of NIC offloading, fairness experiments, and application latency experiments, *SCRR*'s component analysis, and a closer look at the performance of existing *PIFO* implementations.



(a) *IPG < 10µs* (b) *IPG < 50µs* (c) *IPG < 100µs* (d) *IPG < 500µs*

Figure 14: *Burst length distribution for Internet traces using various IPG thresholds.*

D.1 Performance of *PIFO* Approximations

Push-In First-Out abstraction [82] presents an interesting breakthrough in realizing priority packet scheduling, paving the way for maintaining priority lists in hardware that can realize schemes such as *STFQ* [42] and *pfabric* [18]. To overcome the practical roadblocks of deploying *PIFO*, recent works attempt to approximate *PIFO* using admission control on a single queue [89], and using strict-priority FIFOs [17].

D.1.1 CPU utilization in *AIFO* and *SP-PIFO* is high

The CPU utilization profile of *AIFO* is strikingly different from conventional packet schedulers. Fig. 15 shows the CPU utilization of the enqueue and dequeue functions separately for the experiment shown in Fig. 8a (§5.3). The CPU utilization of *AIFO* dequeue function is lower than even Tail-drop (Fig. 15b), because *AIFO* keeps the queue much shorter than other packet schedulers (§D.1.2) and reduces the overall cache footprint. On the other hand, the CPU utilization in enqueue is very high (Fig. 15a) due to the rank sampling and percentile calculation [89]. Since the rank sampling of *AIFO* can be tuned, we configured *AIFO* to sample the rank every 16 packets and to maintain up to 64 samples. We increased the maximum number of samples from the default [89] to improve *AIFO*'s fairness. In practice, the number of samples is usually much lower than 64, because only the samples still in the queue are considered, and the queue is kept short. Changes to the rank sampling rate and the maximum number of rank samples would obviously impact *AIFO*'s fairness and CPU utilization, however we haven't found a configuration giving better overall performance than *STFQ* in our experiments.

The CPU utilization profile of *SP-PIFO* is more balanced. In enqueue, there is additional overhead in choosing the proper priority sub-queue and keeping track of the rank of each priority sub-queue (Fig. 15a). In dequeue, there is additional overhead in scanning the priority sub-queues (Fig. 15b). For the majority of data points, *SP-PIFO* has higher CPU utilization than *STFQ* with higher application latency (Fig. 21, 22, 23).

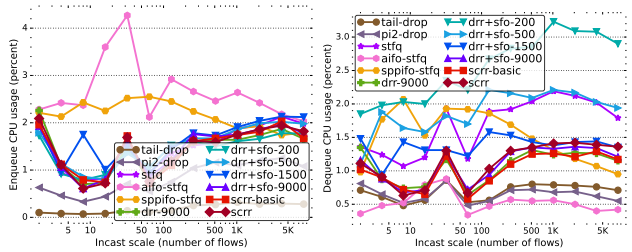
D.1.2 *AIFO* causes queue underutilization

AIFO selectively drops packets prior to admitting them in its single queue. For each packet, it computes a rank, and compares this rank to the distribution of ranks for packet already in the queue. If the percentile of the packet is lower than the current queue occupancy, the packet is dropped, otherwise it is admitted. *AIFO* implements *STFQ* by calculating the packet rank from packet virtual times.

In Fig. 16a, a single TCP flow with increasing RTT is run through a bottleneck. The bottleneck is created using Linux TBF rate limiter [54], while the RTT in increased by adding a fixed delay with Linux *NetEm* [46]. Each scheduler is configured for 15 ms queuing delay. The theory of buffer sizing [46] predicts that the 15 ms queue should be large enough for TCP Cubic traffic with RTT up to 35 ms. This is the case for all schedulers, except *AIFO*.

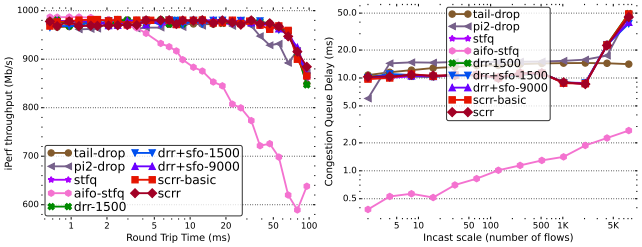
The problem is that when using *STFQ*, the statistical distribution of ranks is highly biased, and the statistical admission control of *AIFO* is no longer appropriate. With a single flow, the virtual time of incoming packets is strictly increasing, as it is always larger than any other packet in the queue, and is therefore always rejected based on its rank. *AIFO* also includes a burst mechanism: when the queue utilization is below 10%, packets are never dropped. As a result, with a single flow, *AIFO* can only use the portion on the queue configured for burst. This underutilization of the queue has direct impact on TCP performance as depicted in Fig 16a.

Fig 16b shows that even for very large number of flows



(a) *LRO - Enqueue* (b) *LRO - Dequeue*

Figure 15: *Impact of increasing Incast scales on CPU utilization in the presence of NIC offloading functions.*



(a) *Throughput as RTT increases* (b) *Queue utilization*

Figure 16: *Throughput and queue utilization of AIFO under different schedulers as RTT and flow count increase.*

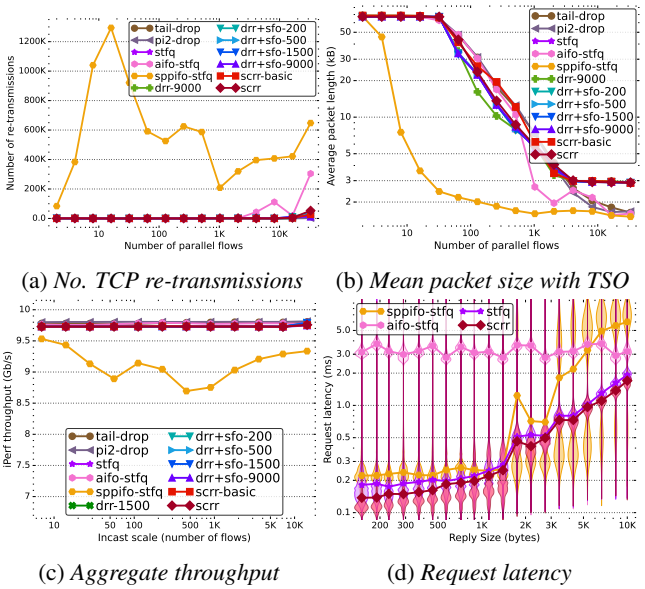
(10k), the statistics of packet virtual times are still sufficiently biased to show severe queue uderutilization with *AIFO*. The amount of queue underutilization decreases with the number of flows, therefore it is not possible to fully compensate for it.

D.1.3 *SP-PIFO* causes packet reordering

We implement *SP-PIFO* using the semantics outlined in [17] and compare its performance against our representative packet schedulers. Fig. 17a presents the TCP re-transmission count for all schemes for the TSO experiment in §5.3. Even with heavy flows congesting the scheduler, *SP-PIFO* suffers from intense packet re-ordering, resulting in unwanted TCP re-transmissions. This in turn impairs the large receive offload functions to save CPU by creating larger packets as depicted in Fig. 17b, ultimately resulting in higher CPU utilization. Hence, *SP-PIFO*'s flow migration across priorities, comes at the cost of heavy re-ordering, starvation, TCP re-transmissions and higher CPU consumption for heavy flows.

Next, we repeat the fairness experiments previously reported in Fig. 9a on a testbed with similar configuration but faster CPUs and report the results in Fig. 17c (§5.4). With faster CPUs, all schedulers except *SP-PIFO* can achieve line rate regardless of the incast scale, even in the case of resource-hungry *DRR+SFO-200*. Similar to the above case, *SP-PIFO* suffers from unwanted TCP re-transmissions that result in reduced throughput.

Fig. 17d presents the request-response workload performance under various packet schedulers; it is an extract of the experiments previously reported in 10a (§5.5). The vertical violins in Fig. 17d represent the distribution of latencies for



(a) *No. TCP re-transmissions* (b) *Mean packet size with TSO*

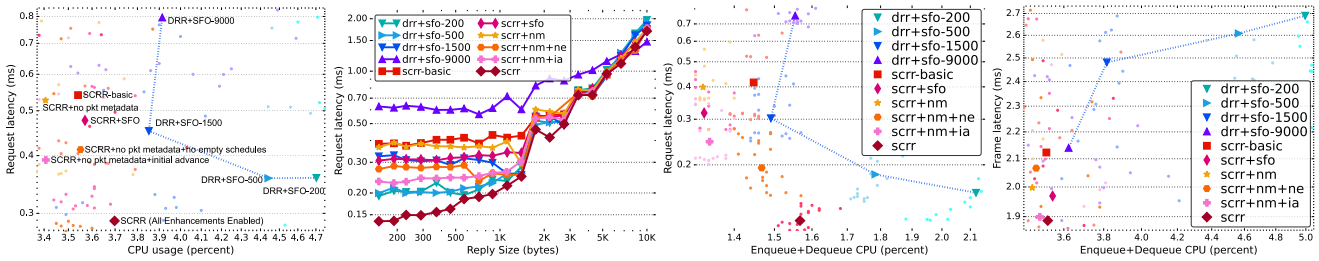
(c) *Aggregate throughput* (d) *Request latency*

Figure 17: *SP-PIFO performance with heavy flows and TSO (top row), under heavy flows (bottom left), and under request-response workloads (bottom right).*

each reply size and each scheduler. Other schedulers have a tight distribution of latency, whereas the latency distribution of *SP-PIFO* is wide and mostly bimodal when the reply size spans multiple packets. Although many requests are lucky to benefit from the highest priority band and are quickly completed with a latency even lower than *STFQ* and *SCRR*, some requests are heavily penalized and lead to poor average performance. This is because, while the first response packet will have a high chance of using the highest priority band, the consecutive packets will have a larger virtual time and will be inserted into lower bands, hence facing starvation.

D.2 *SCRR* Component Analysis

Fig. 18a compares the CPU utilization and response latency of different *SCRR* components for the request-response experiment reported in §5.5. The basic *SCRR* algorithm (without bursty flow enhancements) offers lower CPU utilization compared to DRR alternatives. The *no packet metadata* variant offers even better CPU performance due to imposing less memory pressure. The *No Empty Schedule* enhancement prevents the *Sparse Flow Optimization* in *SCRR* from performing unwanted visits into stale flow entries in the old flows list by immediately kicking the flows out of the schedule and re-inserting them back when they become active. This results in improved responsiveness by lowering the latency by 23% compared to the basic algorithm. The *initial advance* enhancement has a similar effect on sparse flows by allowing them to send a slightly larger burst without violating the fairness bounds. When both enhancements come together in *SCRR*, the best response latency results can be seen, improving the basic algorithm by 46% while performing 18% faster than



(a) *CPU util. vs latency* (b) *Response size vs latency* (c) *Edge: Varying Request Size* (d) *Router: Varying VBR pkt size*

Figure 18: *Comparing the performance of individual SCRR components for the request-response experiment in §5.5.*

DRR+SFO with 200B quantum configuration. Finally, with increasing reply lengths, the upper hand of Sparse Flow Optimization starts to fade as the latency-sensitive flows become backlogged in the scheduler and exhaust their fair share (Fig. 18b). We repeat this experiment on the Edge switching topology and report the results in Fig. 18c in which similar trends in the CPU utilization and response times can be observed for SCRR variants. Finally, Fig. 18d presents the CPU vs latency trade-off for a VBR traffic on the routing topology. SCRR-basic shows a better adaptation to the VBR traffic compared to DRR by offering 15% lower latency and 8% lower CPU utilization with respect to DRR-1500 setting. Other enhancements of SCRR further increase the latency gap by 16%.

D.3 Application Performance Under BBRv3

Request-response Flow Performance. Small request-response flows are one of the most prevalent traffic patterns of modern applications [22, 62, 71]. We already presented experiments with request-response flows using TCP Cubic in §5.5 and §D.6, this section repeats those experiments with BBRv3 and highlight the differences between using BBRv3 and TCP Cubic. We integrated BBRv3 from its official repository [13] and use a different testbed with identical configuration to Fig. 7 with Linux machines running kernel 6.10. The main difference is that on this testbed the scheduler is running on much faster CPUs (Intel Xeon W5-3435X), and therefore the packet schedulers with high CPU usage are less penalized.

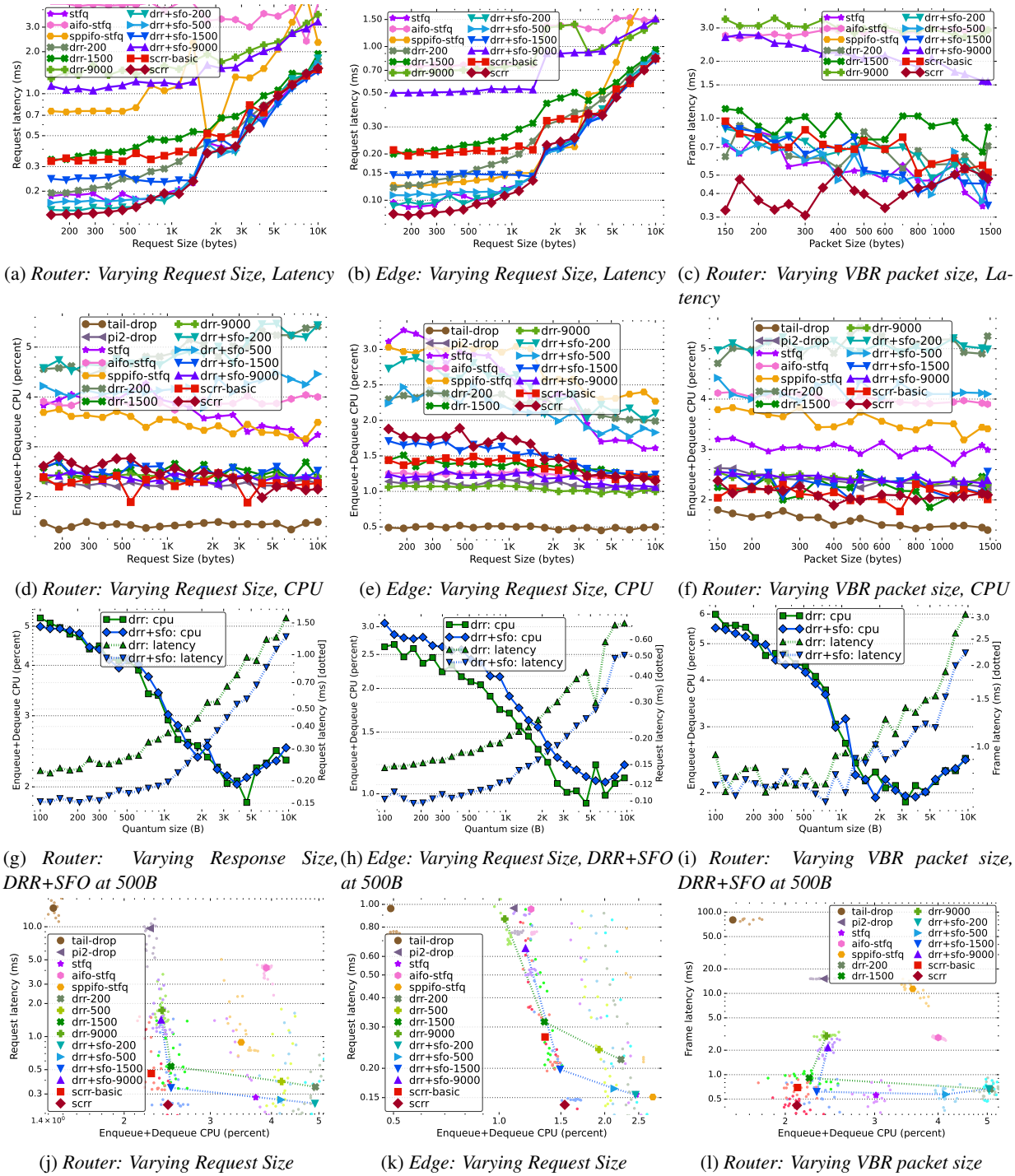
Fig. 19a depicts the application performance as we increase the size of the requests from 100B to 10 kB while the reply size is set to 100B on the router in topology 7b. For this experiment, two sender machines generate 8 latency-sensitive flows each, similar to the Cubic experiments (§5.5). Another two senders generate 64 parallel background long-lived TCP flows each. All flows are configured to use BBRv3. We intentionally opt not to enable Explicit Congestion Notification (ECN) [70] support in BBRv3 as it can arbitrarily impact scheduling performance, and the integration of ECN in *SP-PIFO* and *AIFO* is not trivial and undefined [82, 89].

Fig. 19d shows that with the faster CPU, the CPU utilization of all schedulers is reduced, and has also less jitter. Fig. 19g illustrates that for request-response workloads, the quantum configuration for DRR under BBRv3 offers the usual tradeoff between latency and CPU usage along with the benefits offered by SFO. Finally, BBRv3 offers a tighter control over queue utilization, which causes all single-queue schedulers, i.e., tail-drop, *PI2*, and *AIFO*, to have a smaller average queuing latency. Nonetheless, *SCRR* is able to offer superior latency compared to all alternatives, with low CPU usage (Fig. 19j). *SCRR* has the lowest average latency (0.24 ms), closely followed by *DRR+SFO-200* (0.25 ms), *DRR+SFO-500* (0.27 ms), *STFQ* (0.28 ms) and *DRR+SFO-1500* (0.34 ms).

Figs. 19b, 19e, 19h, and 19k repeat the request-response experiment on the edge topology (7a). Again, the same trends in both application latency and CPU utilization in the face of different request sizes can be observed. Linux's control over the transmit socket buffer size prevents the backlog to exceed 1 ms for tail-drop which significantly improves its application response time. As a result, both *PI2* and *AIFO* do not show any meaningful latency improvements over *Tail-Drop*. Those results are pretty much equal to our results with Cubic (§D.6), indicating again that the congestion control has little effect on overall performance in this setup. *SCRR* has the lowest average latency (0.14 ms), closely followed by *SP-PIFO* (0.15 ms), *DRR+SFO-200* (0.15 ms), *STFQ* (0.15 ms), *DRR+SFO-500* (0.16 ms) and *DRR+SFO-1500* (0.19 ms).

Streaming Flow Performance. The traffic of video conferencing applications is another important traffic pattern of modern applications [52, 61, 79]. We already presented experiments with Variable Bit Rate (VBR) flows using TCP Cubic in §5.5 and more extensively in §D.6, this section repeats those experiments with BBRv3 and highlight the differences between using BBRv3 and TCP Cubic.

VBR experiment results under BBRv3 are depicted in Figs. 19c, 19f, 19i, and 19l. In this experiments, 2 senders generate 32 concurrent VBR UDP flows each, and two other servers generate 64 long-running heavy TCP flows each. Here, the major difference from Cubic experiments (§5.5) is that *DRR*'s overall latency is barely affected when changing the quantum, which is attributed to the faster CPU. The smaller quanta does increase the CPU utilization, and for this experiment, the optimal quantum is around 1500 B. Furthermore, *SFO* improves the latency of *DRR* at little CPU cost. As usual, *SCRR* meaningfully reduces both the latency and the CPU uti-



(a) *Router: Varying Request Size, Latency* (b) *Edge: Varying Request Size, Latency* (c) *Router: Varying VBR packet size, Latency*

(d) *Router: Varying Request Size, CPU* (e) *Edge: Varying Request Size, CPU* (f) *Router: Varying VBR packet size, CPU*

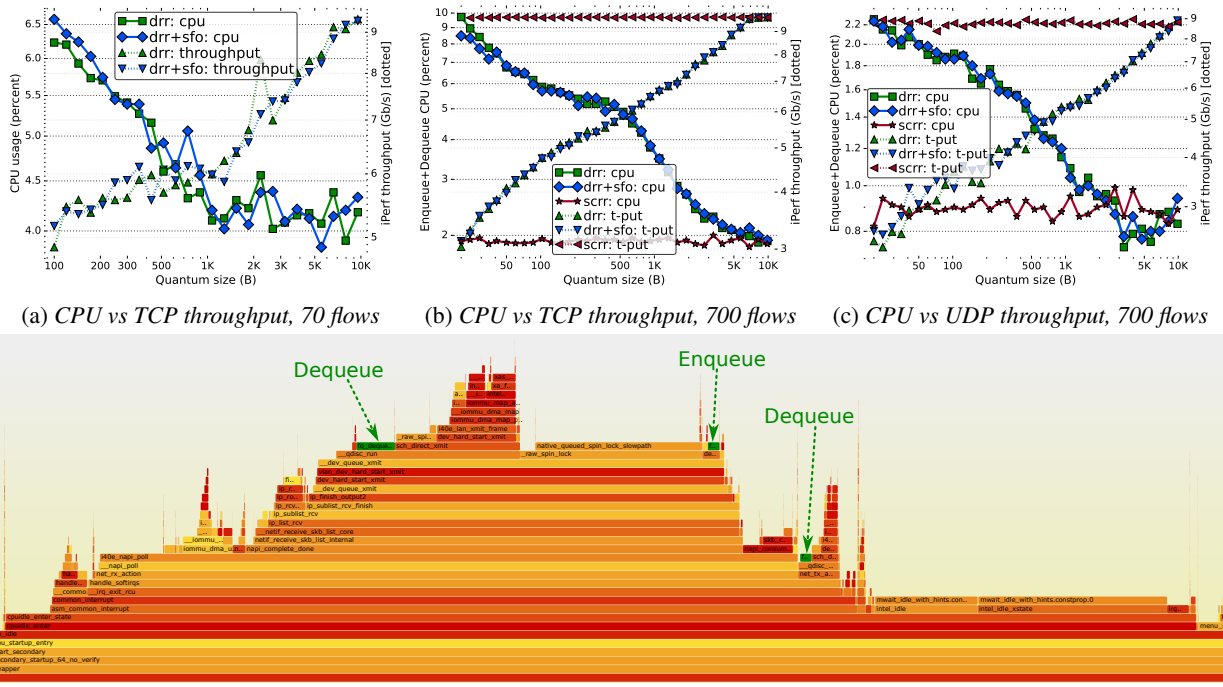
(g) *Router: Varying Response Size, DRR+SFO at 500B* (h) *Edge: Varying Request Size, DRR+SFO at 500B* (i) *Router: Varying VBR packet size, DRR+SFO at 500B*

(j) *Router: Varying Request Size* (k) *Edge: Varying Request Size* (l) *Router: Varying VBR packet size*

Figure 19: *Impact of packet schedulers on application response times for latency sensitive workloads with BBRv3 transport.*

lization. In this experiment, *SCRR* has the lowest average latency (0.42 ms), followed by *STFQ* (0.56 ms), *DRR+SFO-500* (0.57 ms), *DRR+SFO-1500* (0.62 ms), *DRR-200* (0.66 ms), *DRR+SFO-200* (0.67 ms), *SCRR-basic* (0.70) and *DRR-200* (0.71 ms).

We observe that scheduling paradigms follow a similar trend under both Cubic and BBRv3. BBRv3 includes advanced techniques to pace packets, to adjust the sending rate, and to try to keep the latency small at the queue, plus a modified slow-start algorithm [13]. However, the bursty requests are too short to show any significant difference compared to Cubic (§5.5). Our results confirm that unlike packet scheduling, congestion control has limited impact on the performance short bursty flows and *SCRR* can complement widely used In-



(a) *CPU vs TCP throughput, 70 flows* (b) *CPU vs TCP throughput, 700 flows* (c) *CPU vs UDP throughput, 700 flows*

(d) *FlameGraph, TCP, DRR, Quantum 500 B*

Figure 20: *Impact of DRR quantum on CPU overhead and traffic throughput.*

ternet congestion control by providing fairness, low resource consumption and superior application performance for such flows.

D.4 Impact of CPU overhead on throughput

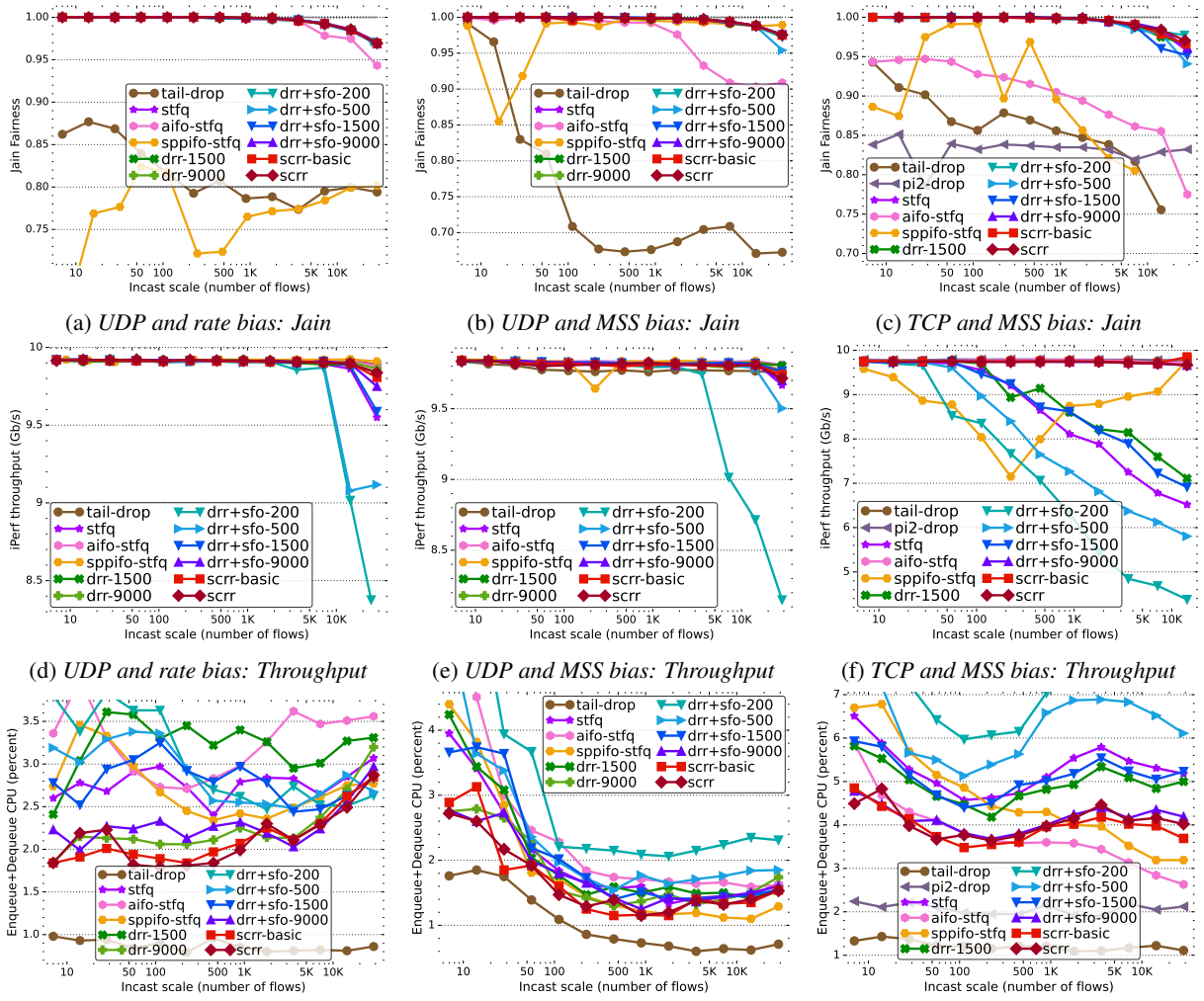
CPU utilization is a good proxy for the overall resource requirements of the packet scheduler. Some software implementations of packet schedulers in Internet gateways result in reduced network throughput due to their high CPU usage [2, 3]. It is usually tricky to estimate the impact of CPU overhead, because different packet schedulers may interact differently with the rest of the networking stack. By changing the quantum of DRR, we can artificially increase the CPU overhead of packet scheduling without changing anything else, and isolate the contribution of the packet scheduler overhead. Further, when the quanta is below 1000 B, packets are scheduled in the exact same order, so only CPU overhead changes. Experiments in Fig. 20 create an incast of heavy flows sharing a bottleneck link serviced by *DRR* and show the increase in CPU overhead due to the reduced the quantum.

Fig. 20a shows the CPU overhead of the packet scheduler and the throughput when using TCP flows. Each of the 7 sender generates 10 flows and each uses a MSS of 1000, 1500, 2000, 3000, 5000, 7000 and 8956 bytes respectively. At higher quanta, the lower per-packet processing time is balanced by the increasing packet rate, and CPU usage is constant. A small 100B quantum makes *DRR* the bottleneck and reduces the aggregate throughput to 5 Gb/s, which is half of the capacity. The CPU usage for *DRR* is less than 7%, however this puts enough pressure on the rest of the networking stack to cause such a slowdown.

Fig. 20b extends this experiment on our new testbed. Each of the 7 senders initiate 100 flows, each using an MSS of 1500, 2000, 3000, 4000, 6000, 8000 and 8956 bytes, respectively. The CPU frequency is downscaled to 1 GHz to mimic the original testbed.[5] The performance of *DRR* is similar to the experiment on the original testbed in Fig. 20a, a small 20B quantum reduces throughput to 3 Gb/s, which is a third of when quantum is 10000B. *SCRR* achieves optimal throughput, however on a slower CPU, *SCRR* could become a bottleneck and cause lower throughput. *SCRR* offers low CPU overhead because every iteration of the dequeue function is guaranteed to dequeue a packet.

Fig. 20c repeats the experiment of Fig. 20b using UDP flows, the flows are constant bit rate and the aggregate rate per sender is 2 Gb/s, for a total of 14 Gb/s. This traffic that the bottleneck can't process is dropped, the ratio of UDP packet drops (not shown) is the inverse mirror of the throughput, a lower throughput means more packets are dropped. The throughput results are broadly similar to the experiment with TCP flows, however the CPU overhead measured is much lower. Also, with the highest quantum, CPU overhead goes up. We repeated these experiments multiple times to eliminate a fluke. We do not have a good explanation for those specific results. The Linux network stack is complex, and cache performance impact and context switch impact can be

[5] In the original testbed, the CPU frequency was not downscaled.



(a) *UDP and rate bias: Jain* (b) *UDP and MSS bias: Jain* (c) *TCP and MSS bias: Jain*

(d) *UDP and rate bias: Throughput* (e) *UDP and MSS bias: Throughput* (f) *TCP and MSS bias: Throughput*

(g) *UDP and rate bias: CPU* (h) *UDP and MSS bias: CPU* (i) *TCP and MSS bias: CPU*

Figure 21: *Fairness of packet schedulers and their CPU overhead as the scale of Incast increases.*

un-intuitive. With UDP traffic, the networking stack has to process a lot more packets and ultimately drop them, causing additional CPU pressure. With TCP traffic, sub-queues may go in and out of the schedule due to the saw-tooth behavior of the TCP congestion control, increasing the amount work for the scheduler. In summary, it's not possible to determine an exact CPU overhead that will cause throughput degradation, this will depend on the traffic and the system.

Fig. 20d shows the the CPU overhead of the various kernel functions in more detail as a FlameGraph for the experiment in Fig. 20b with quantum size 500 B. The call stack for the *enqueue* function starts with the kernel polling Ethernet drivers (__napi_poll), which then passes newly received packet to the networking stack (netif_receive_skb_list_internal), those packets are routed (ip_finish_output2) and then directed to a network device (__dev_queue_xmit) and finally the *enqueue* function is called. Most of the calls to the *dequeue* function are when a packet is directed to a network device (__dev_queue_xmit), effectively after enqueuing some packets, the network stack try to make progress sending packet from that queue. Another call stack for the *dequeue* function is from a periodic timer in the network stack (net_tx_action). As expected, the *dequeue* operations consume more CPU than *enqueue*, because it needs to spin to accumulate quanta for each flow (which is a similar result to Fig. 15). During packet processing many other parts of the Linux network stack must be exercised that respectively consume CPU. With more pressure on the qdisc, the network stack processing also experiences more CPU contention and cache pressure. This explain why even if the qdisc uses a fairly low percentage of the CPU, it can be a bottleneck and lead to throughput decrease.

D.5 Fairness Under Different Biases

The crucial property of a packet scheduler is enforcing fairness and QoS. This section is an extended version of the fairness experiments already shown in §5.4. Again, to better

scale the experiments, we configured all scheduling classes with identical weights and configured various biases between flows to create potential unfairness. These experiments showcase how the scheduler achieves fairness by overcoming these biases.

The first bias we introduce is the rate at which flows arrive at the scheduler. This is a much-generalized version of the experiments in previous papers [81], where it aims to investigate how well a scheduler deals with incast traffic and a large number of misbehaving flows. The testbed is the original testbed (§5.2) and we are using the routing topology 7b. The seven senders generate UDP traffic, each with different aggregate offered rates of 1.5, 2, 3, 4, 5, 6, and 7 Gb/s, respectively. The number of UDP connections per sender varies from 1 to 4096, for a maximum of 28672 flows. Fig. 21a depicts the Jain Index measuring fairness across the flows. The *tail-drop* and *SP-PIFO* schedulers show poor fairness since it admits packets proportional to their arrival rate, favoring UDP flows with higher rates. All fair queueing schedulers, including *SCRR*, have excellent fairness, as expected, as they enforce byte-level fairness for every flow. At a very high number of UDP flows (28k), the fairness of all schedulers is impacted by hash collisions in the classifier, a well-known caveat of using hashing in flow classifiers. Most QoS classifiers have well-defined QoS classes that do not use hashing, and would not suffer from this issue.

Fig. 21d presents the aggregate throughput of the UDP flows. *DRR-200* and *DRR-500* are methods that waste CPU cycles (§2.4) and they result in decreased throughput at higher flow counts. This is because an increase in flow count increases the cache footprint of the scheduler and the probability of cache misses. Fig. 21g presents the CPU overhead on the router. In general, a higher CPU overhead is more likely to impact throughput (§2.4), however, in this experiment, the relation between the two is not obvious. We believe that the need to drop a very large number of packets distorts the CPU overhead numbers measured. A lower quantum increases CPU consumption for *DRR*, as more scheduling cycles are needed to accumulate enough quantum. All variants of *SCRR* offer lower CPU utilization amongst the schedulers.

The second bias we introduce is the packet size. Varying the packet sizes forces the schedulers to transmit more packets from flows that send smaller packets in order to satisfy the proper byte-level accounting of the schedulers. The seven sender machines generate UDP traffic, each using a different Maximum Segment Size of 2500, 3000, 3500, 5000, 6000, 8500, and 8956 bytes, respectively. Fig. 21b shows the Jain Index measuring fairness across the flows. The *tail-drop* queue yields poor fairness again. This is because the incast traffic causes the single FIFO to always be at its peak utilization. In this state, it becomes more probable for the queue to have enough space for small packets under the byte limit (18 MB, or 15 ms of traffic), which further favors flows with small packets. All other schedulers have excellent fairness, as expected. Fig. 21e shows the aggregate throughput of the UDP flows, again *DRR-200* and *DRR-500* show reduced throughput. Fig. 21h shows the CPU overhead on the router, asserting *SCRR*'s low CPU overhead.

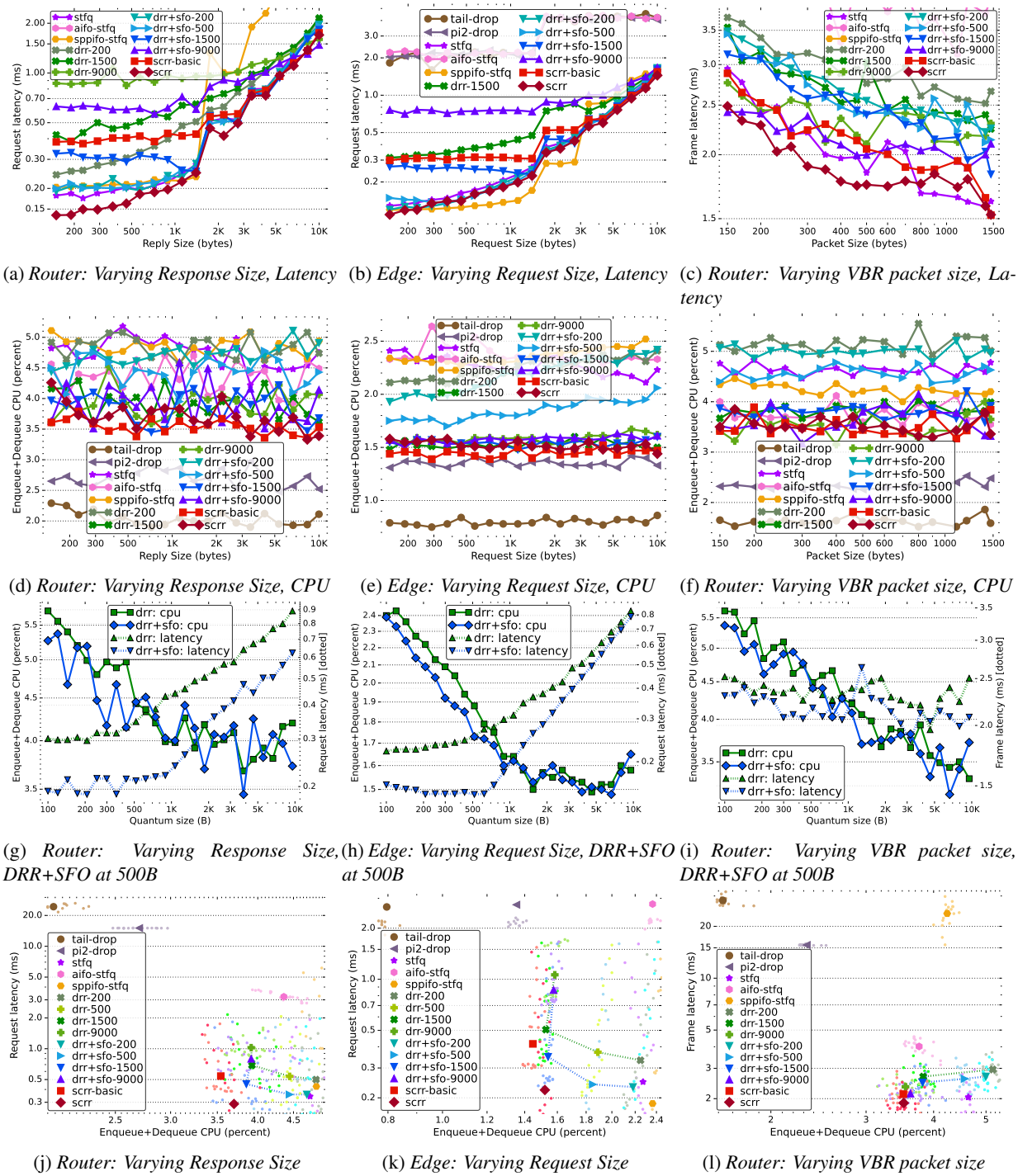
The packet size bias can also be used for TCP traffic. Fig. 21c plots the Jain Index for TCP Cubic flows. At a very high number of TCP flows (28k), some flows experience reduced performance or stalls, especially with schedulers that use the *tail-drop* queue. The TCP congestion protocol is not designed to provide byte-level fairness, therefore *tail-drop* shows only slightly better fairness compared to UDP. However, this time, flows with larger packets are favored. In the congestion avoidance phase, TCP increases the congestion window by one whole packet for each RTT, therefore flows with larger packets can increase their throughput faster. The *PI2* AQM is not designed for byte-level fairness either, which explains why it performs similarly to *tail-drop*. The same applies to *AIFO* and *SP-PIFO*. All fair schedulers provide superior fairness as they enforce byte-level fairness amongst classes. Fig. 21e shows the aggregate throughput of the TCP flows. The TCP flows are much more impacted than UDP flows, as packet drops cause the congestion control to reduce the congestion window. Fig. 21i shows the CPU overhead on the router. In this experiment, the throughput decrease can be directly correlated with the CPU overhead. As expected, the schedulers consume more CPU than the *tail-drop* and *PI2*, and as in previous experiments, the *SCRR* variants are amongst the schedulers with the lowest overhead.

D.6 Application Performance Under Cubic

We present the extended application latency results from §5.5.

Request-response Workloads. Modern networks are filled with small request-response flows [22, 62, 71]. We designed experiments showcasing how schedulers can help those small requests when they are mixed with long-lived heavy flows. Single queues often suffer from BufferBloat, where a few heavy flows can fill the queue and cause high latency for all other flows. It is fairly easy to show how schedulers provide lower latency than a single queue, however, we found it difficult to show the difference between the various schedulers on latency experienced by applications. The main issue is that those differences are mostly visible at very small request sizes, but those small requests magnify the processing overheads and inefficiencies of the packet processing pipelines, especially in the software. Additionally, the difference in scheduling delays is an order of magnitude lower than the latency added by necessary system optimizations such as interrupt coalescence, packet batching and NIC offloads. Consequently, in many cases, those differences are buried in the noise.

Our first experiment mixes small requests with long-lived heavy flows on the router in topology 7b. Two of the receiver machines generate 8 latency-sensitive flows each, the flows



(a) *Router: Varying Response Size, Latency* (b) *Edge: Varying Request Size, Latency* (c) *Router: Varying VBR packet size, Latency*

(d) *Router: Varying Response Size, CPU* (e) *Edge: Varying Request Size, CPU* (f) *Router: Varying VBR packet size, CPU*

(g) *Router: Varying Response Size, DRR+SFO at 500B* (h) *Edge: Varying Request Size, DRR+SFO at 500B* (i) *Router: Varying VBR packet size, DRR+SFO at 500B*

(j) *Router: Varying Response Size* (k) *Edge: Varying Request Size* (l) *Router: Varying VBR packet size*

Figure 22: *Impact of packet schedulers on application response times for latency sensitive workloads with Cubic transport.*

are a sequences of short back-to-back request-response with a request of 100B and varying response lengths. To highlight the difference between schedulers, we need to make them congested, because if there is no packet accumulation, packets are sent as soon as they arrive at the schedulers and they act as a FIFO. To ensure congestion, another two senders generate 32 parallel background long-lived TCP flows each. The requests are initiated on the receivers, so that the replies go from the sender to receiver and get mixed with the background traffic. For all flows, the MSS of TCP set to 1456B (MTU 1500B). LRO is disabled on the software router to minimize the unwanted latency. The latency of the TCP requests and the CPU utilization of the packet scheduler are measured under various request sizes and queue configurations.

Fig. 22a presents the latency of a selected subset of the schedulers. As the size of the replies increases, they are

split across more TCP segments, requiring more scheduling rounds to be forwarded, and we see a corresponding increase in latency. The theoretical optimal quanta for *DRR* is the MTU size, which is 1500B, however *DRR* with smaller quanta can reach the performance of *STFQ*, whereas *DRR-9000* has the worst latency among the schedulers because it always tries to dequeue 9000B worth of packets on every schedule. *SCRR* has a lower latency than *SCRR-basic* due to the combination of *Sparse Flow Optimization* (§3.3.1), *No Empty* (§3.3.3) and *Initial Advance* (§3.3.2) enhancements, which allow, most of the time, to send all the packets of the latency-sensitive flow in one attempt ahead of the background flows.

Sparse Flow Optimization implemented in DRR and SCRR (Algorithm 4) already allows previously idle sub-queues to be scheduled ahead of heavy hitters, giving an advantage to latency-sensitive flows and flows that use less than their fair share bandwidth. The replies are a burst of packets of various length, it usually includes a TCP ack and one or more TCP packets for the reply. The number of packets in the burst depend on the reply size, reply size greater than a MSS need to be broken down. It's also depend on the timing between the TCP ack and the reply. In our experiments, the TCP ack is not deferred into the reply and is sent as a separate packet. The difference in latency between schedulers is due to how they handle those consecutive packets, in particular whether the subsequent packets have to wait for a full scheduling round or not, and how long a scheduling round lasts. *Sparse Flow Optimization* allow those packet train to be scheduled ahead of backlogged flows. *Initial Advance* (§3.3.2 allow SCRR to send multiple of those packets in the same schedule. And *No Empty* (§3.3.3) allows the packets of the train that did not arrive initially to still be scheduled in the current round. Those mechanisms explain the performance of *SCRR*.

Fig. 22d shows the CPU overhead on the router. The SCRR variants are again amongst the schedulers with the lowest overhead. *SCRR* has a slightly higher aggregate CPU utilization than *SCRR-basic* mostly due to higher ratio of small packets processed (increasing the total number of packets) and the *No Empty schedule* (§3.3.3) enhancement, imposing cache pressure by re-inserting sparse flows into the schedule on their arrival.

Fig. 22g presents the performance trade-offs of *DRR* and *DRR+SFO* with 500B response sizes. A smaller quantum enables *DRR* to reduce average latency, however, it causes *DRR* to consume more CPU. In this experiment, 500B looks to be a sweet spot for quantum configuration, offering the best latency improvements.

Finally, Fig. 22j summarizes the average CPU overhead and average latency over all the request sizes. *Tail-drop* and *PI2* use a single queue, therefore, the background TCP flows quickly fill that queue and impose a full queueing delay to the response packets, or more. All fair-queueing schedulers offer much better latency performance by trading off more compute resource usage. *SCRR-basic* and *SCRR* offer the lowest CPU utilization among fair queueing schedulers. *SCRR* and *SCRR-basic* adapt to packet sizes and are therefore able to offer smaller latency than *DRR-1500* while offering a CPU usage even lower than *DRR-9000*. *SCRR-basic* has lower CPU usage than *DRR-1500* mostly because it can decide to schedule the current sub-queue without having to consult the next packet in that sub-queue. *SCRR* has the lowest average latency (0.29 ms), closely followed by *STFQ* (0.34 ms), *DRR+SFO-200* (0.35 ms), *DRR+SFO-500* (0.35 ms) and *DRR+SFO-1500* (0.45 ms).

The bursty flows are using TCP Cubic, however the latency of those flows with per-flow scheduling is mostly independent of the congestion control, and we believe other congestion controls or UDP flows should have similar performance. Our experiments with BBRv3 in §D.3 partially confirm it. With per-flow scheduling, short bursty flows never accumulate enough packets in their sub-queue at the bottleneck to cause congestion and trigger a packet drop (or ECN signal when available). With per-flow schedulers, even under heavy congestion, we did not observe any packet re-transmissions for the request-response flows. Without packet drops, the TCP Cubic never has to throttle the sending rate, and therefore the performance of those short bursty flows is entirely dictated by how they are serviced by the scheduling algorithm at the bottleneck. For similar reasons, Slow-start is not an issue for such small requests, the Initial Window of TCP is usually big enough to send the full request. With *Tail-drop*, we see a very low level of re-transmissions, and some connections that fail to connect entirely. Packets of bursty flows are added at the tail of an already full queue, where they can get dropped, and the congestion control will need to re-transmit them. This is an issue for those short request-response flow : no data is waiting after the missing packet, consequently Fast Recovery [36] usually does not happen and the retry will only happen with the slower Retransmit Time Out (RTO). This explains why with *Tail-drop* in Fig. 22j, some requests have a latency greater than the queuing delay (15 ms). With *PI2*, the number of packet dropped is much lower than *Tail-drop* [73], which helps to reduce latency to the queuing delay.

Our second experiment mixes small requests with long-lived heavy flows on the edge-switching topology 7a. This time, the request are varying in size and mixed with the background traffic to resemble applications such as POST and PUT methods in HTTP, Telemetry, Analytics, and Email which send larger chunks of data as a request. Two of the sender machines generate 8 latency-sensitive flows each, the flows are a sequences of short back-to-back request-response with a varying request length and a 50B reply. Another two senders generate 32 parallel background long-lived TCP flows each. Figs. 22b show similar results to our first experiment (fig. 22a). One big difference is *SP-PIFO*, for requests that fit in a single packet, it offers very low latency. However, *SP-PIFO* is unfair and usually has wider variance of latency when requests span multiple packets (§D.1.3). According to Fig.

22e, the CPU utilization is much more stable than our first experiment (fig. 22d) because the arrival of packets is more deterministic in this topology, it is no impacted by processing at the receiver NIC like in topology 7b. Fig. 22h shows the usual tradeoffs of *DRR* and *DRR+SFO*. Again, 500B seems to be a sweet spot for quantum configuration if latency is preferred. The default quantum of the Linux 'sch_fq' module is 3000B [33], i.e., 2× the MTU, which indicates that it was most likely optimized for low CPU utilization. Finally, 22k collects all previous results. *SP-PIFO* has the lowest average latency (0.19 ms), followed by *SCRR* (0.22 ms), *DRR+SFO-200* (0.23 ms), *DRR+SFO-500* (0.24 ms), *STFQ* (0.25 ms), *DRR-200* (0.33 ms) and *DRR+SFO-1500* (0.34 ms).

Tail-drop and *PI2* offer considerably lower latency than expected and than the first experiment. The reason is that the Linux kernel keeps in check the number of outstanding SKB packet buffers for each socket, in order to prevent queue build ups on sender machines and to reduce the latency. We measure much lower queue size in all schedulers, in *DRR* and *SCRR* the sub-queues for heavy flows are much shorter than in the first experiment by almost a factor 10. However, the mechanism to limit outstanding packet buffers reduces the latency only for *Tail-drop*, *PI2* and *AIFO*, in other schedulers it has little impact on the latency of the requests. The smaller queue size also reduce CPU utilization for all scheduler, mostly through reduction of the CPU cache footprint. This mechanism has also the potential of affecting adversely the performance of the packet schedulers, if the sub-queue of a heavy flow does not refill fast enough, it could lose some transmit opportunity, however we believe this is not happening in this experiment. There are also potential issue of CPU starvation and synchronization. We have seen unexplainable results in this topology, and we believe that the performance of *SP-PIFO* in this experiment is one of them.

Streaming Flow Performance. The previous section explores the very short packet bursts created by applications using request-reply. This section explores the longer burst patterns created by streaming applications. The traffic of video conferencing applications is usually composed of short to medium bursts that can saturate the network [79], each burst encoding a video frame. We use the isochronous support in *iperf* to generate Variable Bit Rate (VBR) traffic [8], each VBR flow is a UDP stream at 1.5 Mb/s, bursting at 30 frame per seconds. This emulates a Zoom video session at 720p [52, 61]. The traffic is Variable Bit Rate (VBR), so the length of the burst of packets varies from one frame to the next. The length of the burst of packets also depends on the packet size, it goes from an average of 333 packets (150 B) to 33 packets (1500 B) per burst. 64 streams of VBR traffic are generated on two senders, they are mixed with 128 long lived heavy flows on two other senders on the router in topology 7b.

Fig. 22c shows the average one way latency to transmit a full frame for a selected subset of the schedulers. The delay are much higher than the previous experiments, the size of the burst of packets (50k) means that multiple scheduling rounds are needed to transmit it, even at the highest quantum of DRR (9k). Conferencing often uses small packets to reduce the impact of packet losses [52, 79]. In this case, using smaller packets decreases performance, the reason is that VBR traffic is sensitive to both latency and throughput, and header and processing overheads of small packets hurt throughput of the burst. Fig. 22f shows that *SCRR* has the lowest CPU utilization amongst per-flow schedulers. Fig. 22i shows that the behavior of *DRR* is the opposite of request-response traffic, here the bigger quanta decrease latency as this traffic is more sensitive to throughput than latency. SFO is still effective and does reduce the latency of *DRR*. Fig. 22l averages the latency and CPU utilization over all packet sizes. *STFQ* maintains low latency despite its high processing overhead. *SCRR* has the lowest average latency (1.89 ms), closely followed by *STFQ* (2.04 ms), *DRR+SFO-9000* (2.14 ms), *DRR-9000* (2.34 ms) and *DRR+SFO-1500* (2.48 ms).

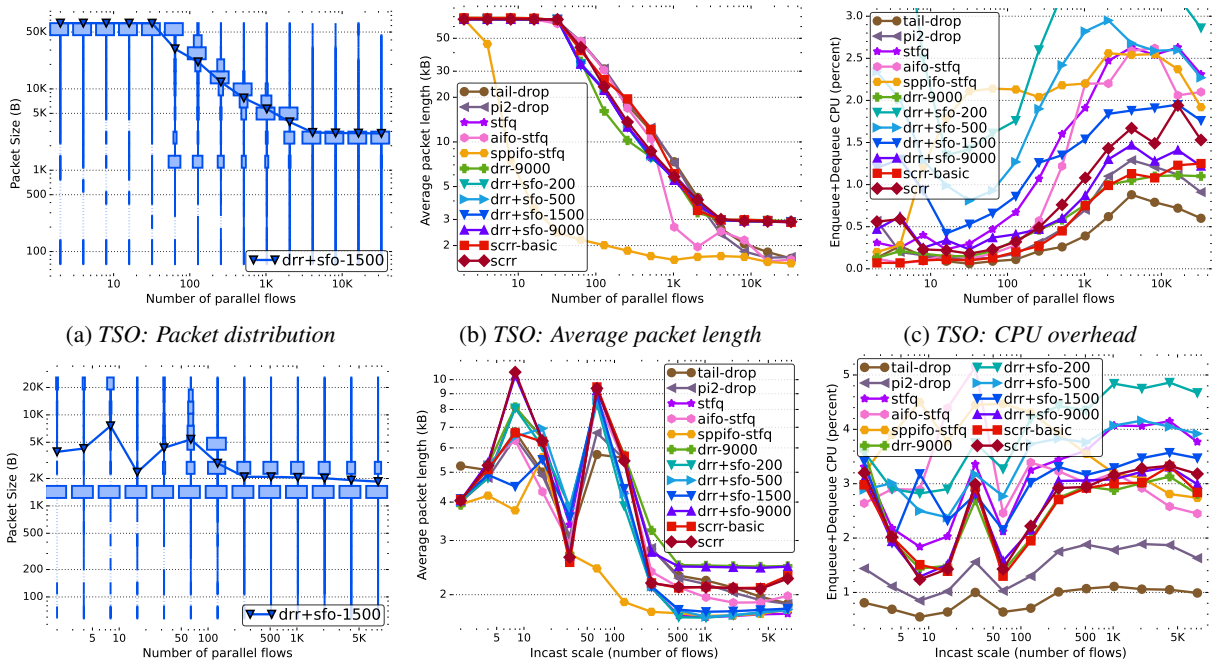
SCRR offers a unique proposition, it offers the lowest latency and lowest CPU usage for VBR frames amongst per-flow schedulers. The latency enhancement of *SCRR* enables the burst to start transmitting sooner, and the low CPU utilization allows to expedite the remaining packets of the bursts and those other packets scheduled ahead of the burst.

D.7 NIC Accelerations - TSO & LRO

We present the extended version of the NIC Accelerations results from §5.3.

The commodity acceleration functions of modern Network Interface Cards (NIC) are essential to sustain the high packet rates for high-speed Ethernet in software. TCP segmentation offloading (TSO) [11] and its software counterpart, Generic Segmentation Offload (GSO), allow the sender to defer the segmentation until the very late packet processing stages, resulting in very large packets being passed to the qdisc. Similarly, TCP Large Receive Offload (LRO) and its software counterpart, Generic Large Receive Offload (GRO), allows a NIC to aggregate received packets into larger chunks. In Linux, the current default maximum size of those packets is 64 kB and there are proposals to greatly increase this maximum [34]. This is because, a reduction in packet rate improves software performance by reducing the overhead of context switches, cache misses, and metadata handling. However, as studied earlier in this work, both TSO and LRO make packet size unpredictable for a software packet scheduler.

Fig. 23a and 23b show the impact of TSO on the packets as seen by the sender host's software scheduler in the edge switching topology (7a). This would be a typical packet distribution seen in a virtual switch deployed in a datacenter or the Cloud, or on a busy server. As the number of flows increases, the individual bandwidth share decreases and the average TSO packet size decreases from 64 kB to 1500B. Fig. 23c



(a) *TSO: Packet distribution* (b) *TSO: Average packet length* (c) *TSO: CPU overhead*

(d) *LRO: Packet distribution* (e) *LRO: Average packet length* (f) *LRO: CPU overhead*

Figure 23: *Performance of packet schedulers under TSO and LRO accelerations.*

presents the CPU overhead of the various schedulers. Smaller packets do increase CPU overhead, as expected. This is a scenario where it is difficult to find an optimal configuration for *DRR*. *DRR-1500* is configured based on the MTU and uses a fixed quantum of 1500B, but this causes too many CPU cycles to be wasted on accumulating quantum over multiple scheduling rounds to send packets that are larger than the quantum. *SCRR* lowers CPU usage by adapting to the packet sizes and has a similar CPU overhead as *DRR-9000* which uses a large quantum.

Figs. 23d and 23e show LRO packet sizes received on the router in topology 7b. This would be a typical packet distribution seen in home routers, SD-WAN gateways, access points or middleboxes. Two sender machines produce half of the workload flows, so the LRO packets are aggregated on two separate NICs. The impact of LRO is a lot less predictable than TSO, but there is still a general trend of higher number of flows resulting in smaller packet sizes. DRR with smaller quantum cannot adapt to the dynamics of the traffic, causing CPU wastage. Again, *SCRR* lowers CPU usage by adapting to the packet sizes and has a similar CPU overhead as *DRR-9000* which uses a large quantum.

The default quantum of the Linux 'sch_fq' module is 3000B [33], i.e., 2× the MTU. From the packet distributions in Fig. 23a and 23d, it's a logical choice as it's a very common packet size, and would allow to send most packets without the need to accumulate deficit for LRO and TSO with high number of flows.