Have your CAKE and eat it too: Scaling software rate limiting across CPU cores

Jonas Köppeler
Technische Universität Berlin
Berlin, Germany
j.koeppeler@tu-berlin.de

Toke Høiland-Jørgensen
Red Hat
Skibby, Denmark
toke@redhat.com

Stefan Schmid
Technische Universität Berlin
Berlin, Germany
stefan.schmid@tu-berlin.de

***Abstract*—Traffic shaping is a critical function for the efficient operation of modern networks, with applications ranging from data center networks to home routers. To correctly fulfill their expected task, traffic shapers must keep up with increasing line rates. While it is possible to scale traffic shapers across multiple CPUs via hardware queues, in some cases — such as when enforcing a global rate limit — these algorithms underperform due to lock contentions. This is especially true within the Linux kernel, where scheduling policies are realized as so-called queuing disciplines (qdiscs) and enforcing a global rate limit can only be achieved on what is effectively a single CPU core.**

In this work, we design and implement a lockless synchronization mechanism that allows qdiscs to efficiently scale rate limiting across multiple hardware queues. To demonstrate its practicality, we integrate this mechanism into the CAKE qdisc, enabling multiple CAKE instances to operate under the MQ qdisc while maintaining a global rate limit. We perform an extensive performance evaluation and find that the implementation achieves close to perfect scaling across cores, with an accuracy deviation with less that 0.25% of the configured rate, while keeping latencies low.

***Index Terms*—Traffic Shaping, Rate Limiting, Kernel, Qdisc**

I. INTRODUCTION

Software rate limiting is a critical technique for ensuring optimal network performance. It is widely applied in various domains, including ISPs enforcing data plans, WAN bandwidth allocation systems [1], and home routers [2].

As line rates continue to increase and surpass CPU speeds, implementing efficient rate limiting becomes increasingly challenging. This is particularly evident in the Linux kernel, where access to queueing disciplines is synchronized through the root qdisc lock, leading to potential contention issues. High lock contention can result in up to 1 second of wait time before a thread is able to enter the critical section, ultimately affecting rate conformance and tail latencies [3]. This issue is apparent in our experiments with a 25G NIC, in which CAKE [2] and HTB [4] were only able to enforce rate limits of up to 8–11 Gbps. The performance even decreased with an increasing number of hardware transmission queues. While there are workarounds for enforcing rate limits on individual traffic classes (such as splitting an HTB tree across transmission queues), it is currently not possible to utilize multiple hardware queues and simultaneously enforce a global rate limit on an interface using the kernel's qdiscs. Further, naively running per-queue rate limiters in parallel without proper synchronization will result in suboptimal rate enforcements [5].

To address these challenges, prior work has focused on overcoming lock contentions and improving the scalability of software rate limiters [3, 6, 7]. One of the most effective solutions is the EDT-BPF approach, which completely eliminates lock contention [7]. This method leverages a BPF program to timestamp packets with departure times before forwarding them to the Fair Queueing (FQ) qdisc [8].

However, in order to maintain low latencies, the EDT model relies on a backpressure mechanism to prevent excessive packet queueing in the network stack [3, 9, 10]. While backpressure can be enforced on end-hosts — the primary target of the EDT-BPF approach — it represents only a subset of rate-limiting applications. Further, using EDT-BPF to enforce a global rate limit on an interface imposes a strict total packet ordering across all FQ instances, effectively eliminating any flow queueing behavior.

We introduce a scalable rate-limiting approach for forwarding devices by implementing a lock-free synchronization mechanism that enables coordination across multiple qdisc instances. We demonstrate its applicability by integrating it into CAKE, enabling enforcement of a global rate limit across multiple hardware queues. The proposed method scales efficiently with increasing CPU core counts while maintaining a deviation of less than 0.25% from the configured target rate. Our design achieves global rate limits up to 3x higher than single-queue CAKE and HTB, while reducing tail latencies by 10x as compared to EDT-BPF. As a contribution to the research community, in order to ensure reproducibility and to facilitate followup research, we make our source code publicly available [11].

The remainder of the paper is organized as follows: Section II details our design of the synchronization mechanism and describes its implementation into the CAKE qdisc. Section III presents a comprehensive evaluation of our solution, highlighting both its performance benefits and limitations. Section IV briefly discusses the findings and elaborates on future work.

II. APPROACH AND IMPLEMENTATION

To evaluate our lockless synchronization mechanism, this work builds on the CAKE qdisc, with a particular focus on its bandwidth shaper. CAKE implements Active Queue

Management (AQM) and rate limiting during packet dequeue, thus overcoming the need for a backpressure mechanism. Through its bandwidth shaper, CAKE enforces a global rate limit on egress network traffic, preventing excessive packet buffering in the lower layers of the network stack. However, CAKE suffers from the aforementioned contention of the root qdisc lock, which prevents it from exploiting the potential of multiple transmission queues.

A. Approach

Our approach overcomes these shortcomings by enabling a scalable version of CAKE to run in combination with the MQ qdisc [12]. This multi-queue version of CAKE is referred to as *mq-cake*. In order to improve scalability and correctly enforce a global rate limit, a synchronization mechanism between *mq-cake* instances is needed. By synchronizing at regular intervals, *mq-cake* instances estimate their local rate limit based on their siblings' activity. This rate estimation is implemented so as to avoid any lock- or atomic operations, thus optimizing scalability. The synchronization frequency determines how fast a qdisc can react to changes in the activity of its sibling qdiscs and comes with a trade-off between accuracy and CPU overhead. The performance implications of the synchronization frequency will be covered in greater detail in section III-C.

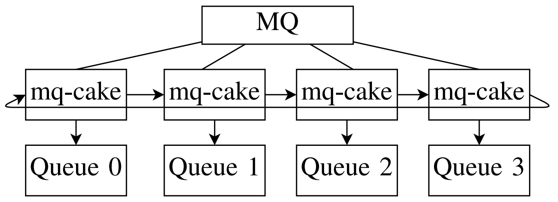


Fig. 1: mq-cake architecture with four transmission queues

B. Implementation

This synchronization mechanism consists of a linked list between all *mq-cake* instances installed on a network interface. Figure 1 shows an example of the proposed architecture for four hardware queues. This approach relies on estimating how many instances of *mq-cake* are actively sending packets — these instances are referred to as either *active queues* or *active qdiscs*. According to this number, a *mq-cake* instance determines its local rate limit by dividing the global rate limit by the number of active *mq-cake* qdiscs.

$$\text{local rate limit} = \frac{\text{global rate limit}}{\text{number of active qdiscs}}$$

Once the qdisc is installed, each *mq-cake* instance scans the list of its siblings in regular intervals to count the number of active qdiscs. The duration of this interval is called the synchronization time, or *synctime*. The default setting for this value is $200\mu s$, which we found to be an appropriate value to ensure fast convergence and low CPU overhead.

A scanning instance considers another qdisc active if it has packets enqueued and/or has sent packets since the last scan. The logic behind these conditions is demonstrated in the following two scenarios: (1) If a qdisc has only large packets enqueued while the global rate limit is low, the inter-packet gap may be larger than the *synctime*. This is due to fact that qdiscs with large packets are slower in their release time and are buffered in the qdisc's queue. This scenario would lead the qdisc to falsely read the instance as inactive if it only considers the *packet sent* condition, as the number of sent packets between the two scans would not have changed. However, any qdisc with packets enqueued is active, as it will eventually dequeue them and use its portion of the configured rate limit. (2) If a qdisc receives very small packets while the global rate limit is high, the backlog of a qdisc consistently remains empty, as packets are less likely to be buffered and thus are immediately sent out. This scenario would also lead the qdisc to falsely read the instance as inactive if it only considers the *has packets backlogged* condition, since the packets are only buffered for a very short interval. Thus, the scanning qdisc needs to maintain a counter — similar to a heartbeat signal — to determine if another qdisc has sent packets since the last synchronization scan.

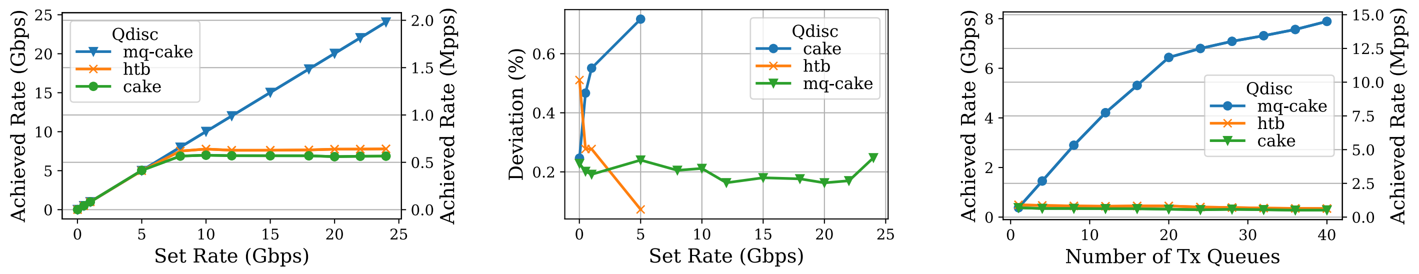
The two activity indicators are read and written non-atomically — however, this does not present a problem, since this approach evaluates the activity indicators in intervals rather than in precise events. In the unlucky event that a qdisc's state changes at the exact moment that another qdisc executes its scan, the change in state will be captured during the next scan. This approach has the benefit of avoiding contention points while still achieving accurate local rate limit estimations. Algorithm 1 shows our proposed synchronization algorithm.

Algorithm 1 Synchronization algorithm

```
 1: procedure GET_ACTIVE_QUEUES
 2:     if now - last_interval < SYNC_INTERVAL then
 3:         return -1;
 4:     end if
 5:     active_queues = 1;
 6:     for all $q$ in qdisc_list do
 7:         if $q$ has packets backlogged then
 8:             active_queues++;
 9:         else if $q$ has sent packets since last interval then
10:             active_queues++;
11:         end if
12:     end for
13:     last_interval = now
14:     return active_queues;
15: end procedure
```

III. Evaluation

In this section, we evaluate *mq-cake* and show its accuracy in enforcing rate limits up to the network card's capabilities of 25 Gbps. *mq-cake* achieves excellent linear scaling with an increasing number of hardware queues and overcomes the



(a) Achieved throughput (b) Relative absolute deviation (c) Scaling behavior

Fig. 2: Rate conformance and deviation for various rate limits (full MTU-sized packets) and scaling behavior (64 byte packets)

limitations of CAKE and HTB. Further, *mq-cake* achieves 10x lower tail latencies as compared to the EDT-BPF approach.

This section is organized as follows: The first subsection describes our experimental setup. Next, we evaluate *mq-cake*'s rate limiting capabilities and the corresponding accuracy as well as its scaling properties in comparison to HTB and single-queue CAKE. The third subsection considers the dynamic properties of *mq-cake*, especially its behavior when the number of active queues and *synctime* change. In the fourth subsection, we test and evaluate *mq-cake* and EDT-BPF under TCP traffic and examine the corresponding latencies. The fifth subsection discusses the limitations of the current approach, particularly the imbalances in loads between queues. Lastly, we summarize the evaluation results.

A. Experimental Setup

The experimental setup consists of two identical servers, both of which are equipped with: (1) an Intel CPU (Intel(R) Xeon(R) Gold 6209U CPU@2.10GHz) with 20 physical cores and hyperthreading capabilities; (2) 192GB RAM; (3) two 25G NICs (Intel XXV710 for 25GbE SFP28 (rev 02)). Both servers run an Ubuntu 22.04.4 LTS with a 6.5.0-35-generic kernel that contains *mq-cake*. These machines are connected back-to-back, where one machine generates traffic and measures throughput either using MoonGen [13] or Flent [14] and the other machine — the Device under Test (DuT) — receives the traffic, enforces a rate limit, and sends the traffic back to the traffic generating device. The generated traffic using MoonGen consists only of UDP flows, with a transmission speed of 25 Gbps. For the Flent tests, 1024 TCP streams are used to saturate the link.

The receiving interface of the DuT distributes the incoming flows in a round-robin fashion across its receive queues. The intel_iommu [15] feature as well as NIC offloading capabilities [16] are disabled on each interface on the DuT. The TIPSY framework [17] is used to orchestrate tests. In order to enforce a global rate limit using HTB, we install a single class at the root-qdisc, which rate limits all network traffic. As mentioned above, the EDT-BPF approach as presented in prior work is not suitable to enforce a global rate limit. To enable a comparison with *mq-cake*, we modify the BPF program implementation to enforce such a global rate limit (Algorithm 2).

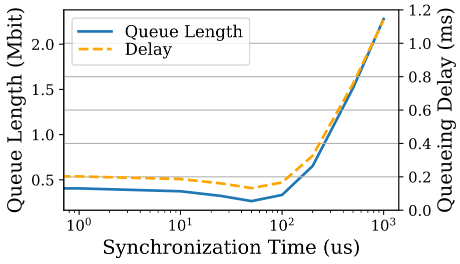
Algorithm 2 EDT-BPF implementation

```
procedure RATE_LIMIT(skb)
    pkt_len = skb→len + compensation
    delay_ns = pkt_len*NS_PER_SEC/global_rate_limit
    next_tstamp = global_next_tstamp

    if next_tstamp ≤ now then
        global_next_tstamp = now + delay_ns
        return TC_ACT_OK
    end if

    if next_tstamp−now ≥ DROP_HORIZON then
        return TC_ACT_SHOT
    end if

    skb→tstamp = next_tstamp
    __sync_fetch_and_add(global_next_tstamp, delay_ns)
end procedure
```

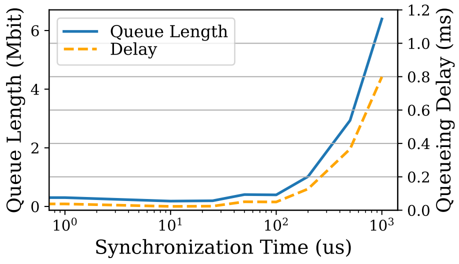
B. Accuracy and Scalability

The most pressing questions about the presented approach are: How accurately does it enforce the configured rate limit? And how does it scale with an increasing number of hardware queues? In this section, we present an in-depth analysis of *mq-cake*'s performance using an unresponsive UDP traffic flood and compare it to the single-queue CAKE and HTB. We demonstrate that *mq-cake* not only achieves excellent rate conformance but also exhibits near-perfect scaling properties.

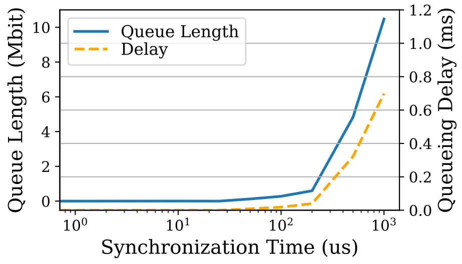
Figure 2a shows the achieved throughput for varying rate limits, ranging from 10 Mbps to 24 Gbps. In this experiment, the network traffic consists of 120 UDP flows containing only full MTU-sized packets. The number of receive and transmission queues is set to 40, meaning that every available logical CPU is assigned one receive and one transmission queue. These settings maximize the achievable throughput and reduce concurrent access to the same qdisc by distributing packet handling across the per-transmission queue qdisc instances. Figure 2b highlights the relative deviation from the configured maximum rate limit. Taken together, Figure 2a and Figure 2b demonstrate that *mq-cake* is able to shape traffic up to 24 Gbps,



(a) Rate limit 2 Gbps.



(b) Rate limit 8 Gbps.



(c) Rate limit 15 Gbps.

Fig. 3: Induced queue lengths and delays at varying *synctimes* and at a configured global rate limit of 2, 8, and 15 Gbps

with a maximum deviation of less than 0.25%. In comparison, HTB and single-queue CAKE plateau at around 7–8 Gbps.

To show the scalability traits of *mq-cake*, we configure the next experiment (Figure 2c) with a rate limit of 20 Gbps and reduce the UDP packet sizes to 64 bytes. Further, we ensure that the number of receive queues always matches the number of transmission queues, preventing imbalances in the traffic load between qdiscs. The effect of these imbalances are further explained in section III-E.

Figure 2c reveals the throughput achieved by *mq-cake*, HTB, and single-queue CAKE in relation to the number of available hardware queues. The experiment shows that both HTB's and single-queue CAKE's performance degrades as more hardware queues become available. This is due to lock contention, which increases as the number of receive queues grows. Under these conditions, an increasing number of CPUs attempt to access the qdisc, which then increases the overall wait time to acquire the root lock. *mq-cake*, on the other hand, scales linearly — the achieved throughput increases at a quicker rate up to 20 transmission queues, after which point the improvement reduces due to the use of hyperthreading cores. This effect is caused by resource-sharing between the two logical cores residing in one physical core: thus, their performance is not completely independent from one another. However, even with hyperthreading enabled, *mq-cake* is still able to increase the throughput.

C. Impact of Synctime

Up to this point, the traffic in the previous evaluations has been held static, meaning that the number of flows, and thus the number of active queues, did not change. With our next experiment, we show and evaluate the impact of the synctime on the rate limiter's accuracy, particularly when the number of active queues changes. To gain insights in *mq-cake*'s accuracy and responsiveness, we induce a change in the number of active queues by increasing the UDP traffic from an initial 4 flows to 40 flows.

Figure 4 shows such a switching event at around 4.94s. During the switch, the throughput spikes due to *mq-cake*'s inaccurate estimation of the number of active queues. Before the switch, only 4 queues were active; during the switch, the remaining 36 inactive queues are activated and then scan

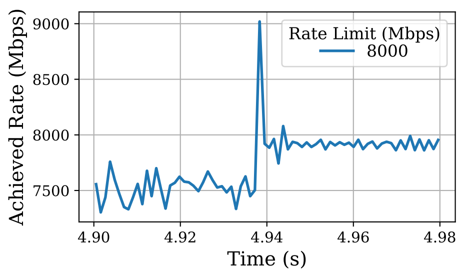


Fig. 4: *mq-cake*'s behavior when switching from 4 to 40 flows with a 200μs synctime

all other qdiscs to estimate their local rate limit. Since this scanning is not necessarily executed simultaneously, the active queue estimation per qdisc will likely be lower than 40 — not all qdiscs will have already enqueued or transmitted a packet at the point of scanning. Further, the 4 already-active qdiscs will not immediately update their estimated rate upon new flow arrivals, which can delay their local rate limit reduction. These conditions result in the observed overshoot in Figure 4.

When evaluating accuracy, it is important to consider the induced queue length at the next bottleneck in the packet path, which is caused by the throughput spike, as well as the increased latencies it produces. The width of the spike can be controlled by manipulating the *synctime*. Exceeding the global rate limit leads to buffering packets in the next device that is in control of a bottleneck link, which ultimately leads to increased latencies and packet drops. To gain insights into the amount of induced latencies and to provision buffer sizes, the next step is to examine these metrics in relation to the *synctime*. Figure 3 outlines the induced queue lengths as well as the corresponding induced queueing delays at three different global rate limits. *Synctimes* beyond 100μs increase the spike's overshoot as well as its duration for the reasons described above. The longer the *synctime*, the longer queues will send an inordinately high number of bytes due to their inaccurate local rate estimation. These plots show that reducing the *syncime* also inhibits the spike intensity as well as the queueing delay.

However, if the *synctime* is too greatly reduced (i.e., less than 50μs in the conducted experiments), the overhead of

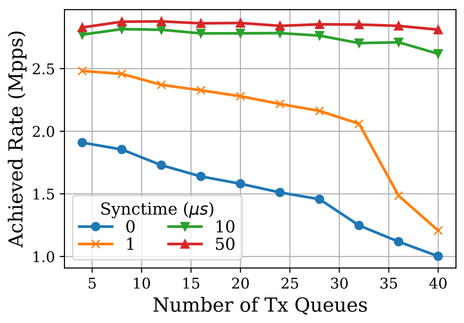
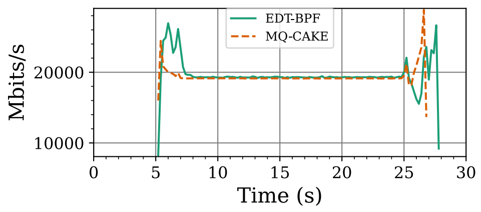
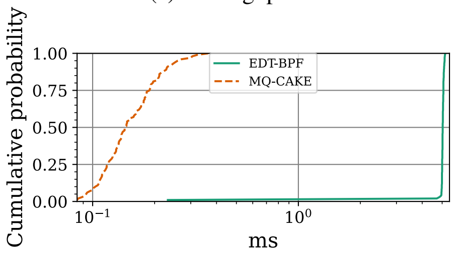


Fig. 5: Achieved packet rate based on the available transmission queues for varying *synctimes*. The traffic consists of 4 flows containing only 64 byte packets. In case of a *synctime* of $0us$, the rate estimation is done for every packet.



(a) Throughput



(b) Ping

Fig. 6: Flent tcp_nup test with 1024 TCP streams, a configured rate limit of 20 Gbps, and a $5ms$ drop horizon for EDT-BPF

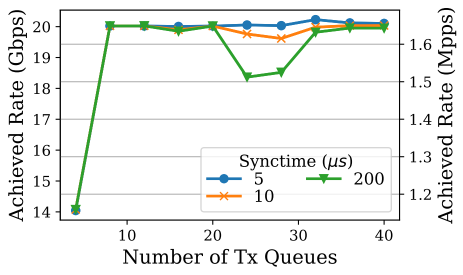


Fig. 7: Achieved throughput in relation to the number of available hardware transmission queues with different *synctimes* for flows with full MTU-sized packets, where the rate limit is set to 20 Gbps and the number of receive queues is held at 40

the synchronization loop increases, lowering the achieved throughput. Figure 5 shows the relation between the achieved rate and the number of transmission queues for different *synctimes*. This plot clearly shows that when the *synctime* is too low, the achieved packet rate decreases due to the synchronization overhead. A greater number of transmission queues increases the *mq-cake* instances' scanning time and may well lead to cache misses when accessing the other qdiscs' activity indicators.

D. TCP and Latencies

The previous experiments are based on unresponsive UDP traffic. To review how these approaches perform with a packet-loss sensitive transport protocol, we inspect the rate conformance under TCP traffic and the resulting latencies using the network testing tool Flent [14] and the TCP Cubic algorithm [18]. Figure 6 compares the performance of *mq-cake* and EDT-BPF under a global rate limit of 20 Gbps.

mq-cake as well as EDT-BPF maintain stable rate enforcement, remaining slightly below the configured limit (Figure 6a). Figure 6b shows the latencies measured during the test execution. *mq-cake* achieves $0.4ms$ latencies at the 99^{th} percentile, a 10x improvement as compared to EDT-BPF. The drop horizon of EDT-BPF directly corresponds with the expected latencies. Even with this configuration, *mq-cake* achieves 10x lower latencies as compared to EDT-BPF.

These experiments highlight *mq-cake*'s ability to maintain low latencies without the need of a backpressure mechanism, thus making it suitable not only for end hosts but also in packet forwarding use cases.

E. Limitations

Over the course of the evaluation, we have shown that *mq-cake* scales excellently with an increasing number of hardware queues while maintaining low latencies. However, we identified that the current approach is less accurate when network traffic is suboptimally distributed across the *mq-cake* instances. So far, our experimental setup ensures that the qdisc layer of the Linux kernel is saturated with packets, which might not always be the case in real world scenarios. In the worst case, this can lead to imbalances between the loads of different *mq-cake* instances, where traffic enqueued in one qdisc cannot saturate the estimated local rate limit while another qdisc instance is heavily flooded with packets. These imbalances taint the active queue estimation and lead to much lower throughput.

Figure 7 shows such an imbalance scenario. In this experiment, the number of receive queues is held at 40 as the number of transmission queues increases. Concentrating first on a *synctime* of $200\mu s$, this plot shows that the achieved rate worsens when the number of transmission queues surpasses half the number of receive queues. At this critical juncture, the traffic is distributed in a way that leaves some transmission queues undersaturated. For example, when there are 24 transmission queues, 16 transmission queues receive double the amount of packets as compared to the remaining 8 transmission queues. The estimated rate of the 8 transmission queues is higher than the traffic they receive, leading to unused bandwidth and a declining throughput. However, as more transmission queues are added, the imbalance is reduced and the per-qdisc rate limit decreases, leading to less unused bandwidth. Such imbalances in multi-queue networking environments are well known in the literature [5, 19].

IV. Conclusion and Future Work

In this work, we present and evaluate a scalable, lock-less synchronization mechanism that allows for the correct enforcement of a global rate limit when scaling to multiple hardware queues. We integrate this synchronization mechanism into the CAKE queueing discipline, thus enabling CAKE to run in combination with the MQ qdisc. We show that *mq-cake* overcomes the scaling limitations of HTB and CAKE while achieving an accuracy deviation of less than 0.25% across a variety of rate limits up to 25 Gbps. Further, *mq-cake* reduces tail latencies up to 10x as compared to EDT-BPF.

Looking ahead, we aim to further explore solutions to address load imbalances between *mq-cake* instances as well as approaches to mitigate overshooting above the configured rate limit during switching events. In addition, we seek to investigate automated approaches to adjust and configure the *synctime* interval. Furthermore, to deepen our understanding of *mq-cake*'s applicability, we plan to evaluate its performance with higher-speed network cards and test it under real-world traffic conditions. There is also potential to investigate other applications that could benefit from our proposed synchronization mechanisms. Additionally, we plan to upstream an API to the Linux kernel, which enables all qdiscs to share state and lays the foundation for a general interface on inter-qdisc synchronization.

Acknowledgement

This project has received funding from the German Research Foundation (DFG), grant 470029389 (FlexNets).

References

[1] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat, "Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 1–14, aug 2015. [Online]. Available: https://doi.org/10.1145/2829988.2787478

[2] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of cake: A comprehensive queue management solution for home gateways," in *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2018, pp. 37–42.

[3] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 404–417. [Online]. Available: https://doi.org/10.1145/3098822.3098852

[4] M. Devera. (2002) Hierachical token bucket theory. [Online]. Available: http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm

[5] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 33–46. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/stephens

[6] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "EyeQ: Practical network performance isolation at the edge," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 297–311. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar

[7] S. Fomichev, E. Dumazet, W. de Bruijn, V. Dumitrescu, B. Sommerfeld, and P. Oskolkov, "Replacing htb with edt and bpf," 2020. [Online]. Available: https://netdevconf.info/0x14/session.html?talk-replacing-HTB-with-EDT-and-BPF

[8] E. Dumazet. (2013) pkt_sched: fq: Fair queue packet scheudler. [Online]. Available: https://lwn.net/Articles/564825/

[9] (2023) Cfp: Bandwidth manager with fq_codel. [Online]. Available: https://github.com/cilium/cilium/issues/29083

[10] T. Høiland-Jørgensen. (2023) The big fifo in the cloud. [Online]. Available: https://blog.tohojo.dk/2023/12/the-big-fifo-in-the-cloud.html

[11] J. Köppeler, T. Høiland-Jørgensen, and S. Schmid, "mq-cake: Scaling software rate limiting across cpu cores," Presentation without proceedings at Netdevconf 0x19, 2025, https://netdevconf.info/0x19/sessions/talk/mq-cake-scaling-software-rate-limiting-across-cpu-cores.html. [Online]. Available: https://github.com/mq-cake/linux-mq-cake

[12] P. McHardy. (2009) net_sched 00/07: classful multiqueue dummy scheduler. [Online]. Available: https://lwn.net/Articles/351021/

[13] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[14] T. Høiland-Jørgensen, C. A. Grazia, P. Hurtig, and A. Brunstrom, "Flent: The flexible network tester," in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 120–125. [Online]. Available: https://doi.org/10.1145/3150928.3150957

[15] Linux iommu support. [Online]. Available: https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt

[16] Segmentation offloads. [Online]. Available: https://docs.kernel.org/networking/segmentation-offloads.html

[17] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári, "The price for programmability in the software data plane: The vendor perspective," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2621–2630, 2018.

[18] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks," RFC 8312, Feb. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8312

[19] B. Stephens, A. Singhvi, A. Akella, and M. Swift, "Titan: Fair packet scheduling for commodity multiqueue NICs," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 431–444. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/stephens