mq-cake: Scaling software rate limiting across CPU cores

Jonas Köppeler
TU Berlin
j.koeppeler@tu-berlin.de

Toke Høiland-Jørgensen
Red Hat
toke@redhat.com

Stefan Schmid
TU Berlin
stefan.schmid@tu-berlin.de

Abstract

Software rate limiting (such as that implemented in sch_tbf, sch_htb and sch_cake) relies on the global qdisc lock to synchronise state, and thus does not scale across CPU cores. This makes it challenging to rate limit at higher rates, since single-core performance has not kept up with network speeds. While there are workarounds for enforcing rate limits on individual traffic classes (such as splitting an HTB tree across TXQs), it is not currently possible to take advantage of multiple hardware queues and still enforce a global rate limit on the interface, using the kernel's qdiscs.

In this work, we implement a multi-queue variant of sch_cake that can scale its rate limiting across hardware queues (and thus CPU cores). We implement this by adding a small bit of shared state across multiple sch_cake instances installed under the mq qdisc. This allows most of the qdisc logic to run under separate per-TXQ qdisc locks, while still supporting a global rate limit for the whole interface. We perform an extensive performance evaluation and show that the implementation achieves close to perfect scaling across cores, with an accuracy deviation of less than 0.25% of the configured rate.

Keywords

rate limiting, qdisc, cake

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 [12], and home routers [6].

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 global qdisc lock, leading to potential contention issues. To illustrate this issue, in our experiments with a 25G NIC, sch_cake and sch_htb were only able to enforce a global rate limit of up to 8–11 Gbps. The performance even decreased with an increasing number of transmission queues.

To address these challenges, prior work has focused on overcoming lock contentions and improving the scalability of software rate limiters [4, 11, 20]. One of the most effective solutions is the EDT-BPF approach, presented at Netdevconf 0x14 by Fomichev et al. [4], which completely eliminates lock contention. This method leverages an BPF program to timestamp packets with departure times before forwarding them to the FQ qdisc.

However, the EDT model decides a packet's fate before enqueuing it in FQ, which makes it impossible to effectively apply AQM algorithms such as CoDel [17]. In order to maintain low latencies, the EDT model relies on a backpressure mechanism to prevent excessive packet queueing in the network stack [20]. The absence of proper backpressure can lead to performance issues, as observed in the Cilium bandwidth manager, where it caused spikes in network latency [2, 7]. 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. In scenarios where direct control over end-hosts is unavailable, such as rate limiting on routers or switches, backpressure mechanisms are not feasible. This limitation is particularly relevant for enforcing data plans at ISPs or managing traffic on home routers. Furthermore, using EDT-BPF to enforce a global rate limit on an interface will create a single virtual FIFO 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 between per-queue sch_cake instances. 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 between 10x–2500x as compared to EDT-BPF. The source code is published on Github: `https://github.com/mq-cake/linux-mq-cake`.

Approach and Implementation

This work builds upon the CAKE (sch_cake) qdisc, with a particular focus on its bandwidth shaper. CAKE implements Active Queue Management (AQM) and rate limiting during packet dequeue, effectively eliminating 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.

Approach

Our approach overcomes the aforementioned shortcomings by enabling a scalable version of *sch_cake* [6] to run in combination with the MQ qdisc [14]. 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 installed *mq-cake* instances is necessary. By synchronizing at regular intervals, *mq-cake* instances estimate their local rate limit based on their sibling *mq-cake* instances' activity. This rate estimation is implemented in a way that avoids any lock- or atomic operations as a means of achieving optimal scalability.

The synchronization frequency determines how fast a qdisc can react to changes in the activity of its sibling qdiscs. If this frequency is higher, the local rate limit is able to converge more quickly to the correct value but the CPU load also increases. When this frequency is lower, the CPU load decreases but takes longer to converge. Further, the CPU load increases as the number of hardware queues grows, since more data structures must be accessed in order to estimate the local rate. However, if the per-packet CPU load becomes too high, the achieved throughput lowers, potentially hindering the system from driving higher rate limits. This phenomenon will be covered in greater detail in the *Evaluation* section.

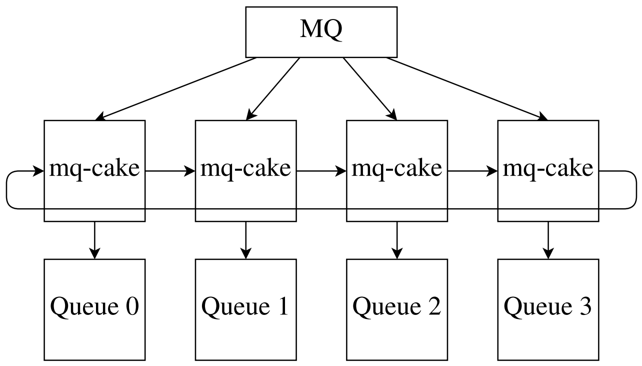


Figure 1: mq-cake architecture with four hardware transmission queues

Implementation

This synchronization mechanism consists of a linked list between all *mq-cake* instances installed on a network interface. During the installation process of a new *mq-cake* instance, this new instance searches for sibling qdiscs and adds itself to their linked list. 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.

$$\text{local rate limit} = \frac{\text{global rate limit}}{\text{number of active qdiscs}} \quad (1)$$

Once the qdisc is installed, each *mq-cake* instance scans the list of siblings in regular intervals to count the number

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
```

of active qdiscs. This duration of this interval is called the synchronization time, or *synctime* for short. The default setting for this value is $200\mu s$, which we found is 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 only considering 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 only considering 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.

This external qdisc observation is necessary because a qdisc cannot accurately determine if it is active or inactive. For example, in cases where packets are immediately sent out, the qdisc would oscillate between active and inactive, leading to inaccurate values for other qdiscs.

The two activity indicators are read and written using the READ/WRITE_ONCE macros. Thus, the reads and writes are racy — 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 would change exactly at the same time as another qdisc executes its scan, the change in state will be captured in the worst case during the next scan. This approach has the benefit of not building contention points while still achieving accurate local rate limit estimations. Algorithm 1 shows this synchronization algorithm.

Evaluation

In this section we evaluate *mq-cake* and show its capabilities of accurately enforcing rate limits up to the network card's capabilities of 25 Gbps and achieves excellent linear scaling with an increasing number of hardware queues. Further, *mq-cake* achieves 10–2500X lower tail-latencies as compared to EDT-BPF approach.

This section is organized as follows: The first subsection describes the 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. In the third subsection we test and evaluate *mq-cake* and EDT-BPF under TCP traffic as well as the corresponding latencies using the Flent [5] network testing tool. The fourth subsection considers the dynamic properties of *mq-cake*, especially its behavior when the number of active queues and *synctime* change. The fifth subsection discusses the limitations of the current approach, especially focusing on imbalances in load between queues. Lastly, we summarize the evaluation results and outline further directions for future works in this field.

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 [3] or Flent [5] 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. In case of the Flent tests, 1024 TCP streams are used to saturate the link. If not mentioned otherwise, hyperthreading is enabled on the DuT.

The receiving interface of the DuT distributes the incoming flows in a round-robin fashion across its receive queues. The interrupts of each receive queue are mapped to exactly one CPU core and irqbalance daemon [9] is disabled. To avoid any side effects from existing firewall and routing rules, the ingress and egress interface of the DuT are attached to a separate network namespace [16]. The intel_iommu [8] feature is also explicitly disabled, since it is enabled by default in the 6.5 Linux kernel version and massively degrades the performance of network IO operations. Further, NIC offloading capabilities like GRO, GSO and TSO [18] are disabled on each interface on the DuT. The TIPSY framework [13] is used to orchestrate tests. For configuring and installing *mq-cake*, the *tc* command line tool is extended [15]. Further, in case of the MoonGen UDP flood tests, the installed *mq-cake* instances are configured with *besteffort flows overhead 18 mpu 64 noatm* [1]. The same options are used for single-queue CAKE. The overhead compensation is configured so that the rate calculation of MoonGen and *mq-cake*/single-queue CAKE are identical. 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 and incorporate the aforementioned overhead compensation. The drop-horizon is set to $2s$, following the value proposed by the authors [4]. The adjusted BPF program is detailed in Algorithm 2.

Algorithm 2 EDT-BPF implementation

```
 1: procedure RATE_LIMIT(skb)
 2:     pkt_len = skb→len + compensation
 3:     delay_ns = pkt_len*NS_PER_SEC/global_rate_limit
 4:     next_tstamp = global_next_tstamp
 5:
 6:     if next_tstamp ≤ now then
 7:         global_next_tstamp = now + delay_ns
 8:         return TC_ACT_OK
 9:     end if
10:
11:     if next_tstamp−now ≥ DROP_HORIZON then
12:         return TC_ACT_SHOT
13:     end if
14:
15:     skb→tstamp = next_tstamp
16:     __sync_fetch_and_add(global_next_tstamp, delay_ns)
17: end procedure
```

To align with the expected buffer occupation, the limit and flow_limit parameters of the FQ instances are configured based on the following formula:

$$\text{Limit} = \frac{\text{Rate Limit} \cdot \text{Drop Horizon}}{\text{\# FQ instances} \cdot \text{MTU size} \cdot 8}$$

If the FQ buffer sizes are not properly adjusted, FQ may drop packets unnoticed by the BPF program, ultimately impacting performance.

Accuracy and Scalability

The most pressing questions about the presented approach is: 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 sch_cake and sch_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 test run, 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

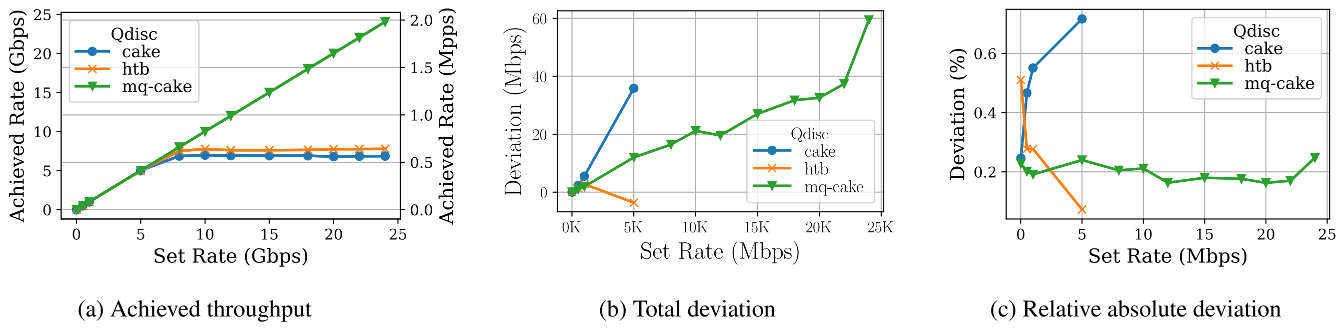


Figure 2: Achieved throughput and deviation from the target rate at various rate limits under network traffic containing only full MTU-sized packets

is assigned one receive and 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 total deviation from the configured maximum rate limit as well as a relative percentage of the rate limit (Figure 2c). Together, these plots demonstrate that both *mq-cake* is able to shape traffic up to 24 Gbps, with a maximum deviation of less than 0.25%. HTB and single-queue CAKE, on the other hand, plateau at around 7–8 Gbps.

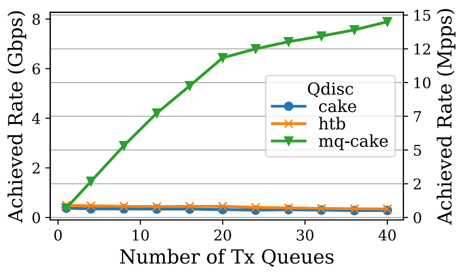


Figure 3: Achieved throughput in relation to the number of available hardware queues for 64 byte packets and a 20 Gbps rate limit

To show the scalability traits of *mq-cake*, we configure the next experiment 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 the Limitations section.

Figure 3 reveals the throughput achieved by *mq-cake*, HTB, and single-queue CAKE in relation to the number of available hardware queues. The test 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 the use of hyperthreading cores. This effect is due to 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.

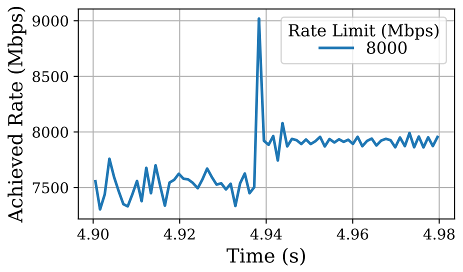
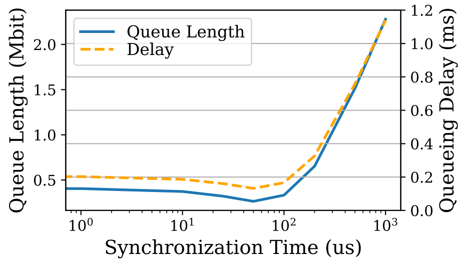


Figure 4: *mq-cake*'s behavior when switching from 4 to 40 flows with a $200\mu s$ synchronization time, full MTU-sized packets, and 40 transmission queues

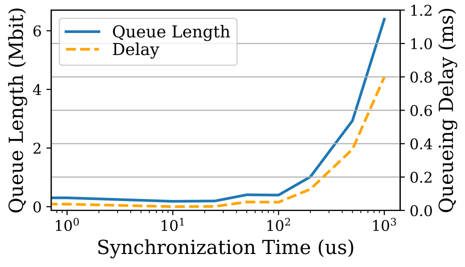
Impact of Synctime

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 initial 4 flows to 40 flows.

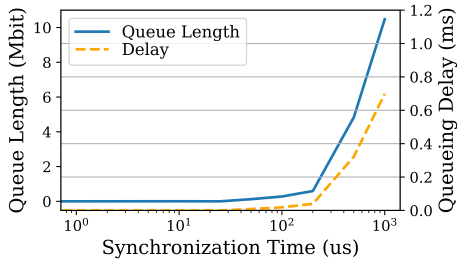
Figure 4 shows such a switching event at around 4.94s. During the switch, the throughput spikes due to *mq-cake*'s inaccurate estimation of the number of active queues. Before the switch, only 4 queues were active; during the switch, the remaining 36 inactive queues are activated and then scan all other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



(a) Rate limit 2 Gbps.



(b) Rate limit 8 Gbps.



(c) Rate limit 15 Gbps.

Figure 5: Induced queue lengths and delays at varying *synctimes* and at a configured global rate limit of 2, 8, and 15 Gbps

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.

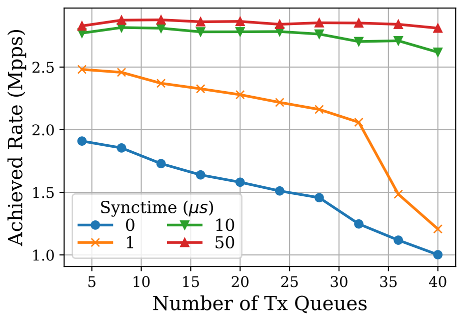
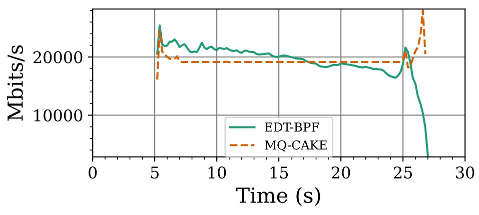
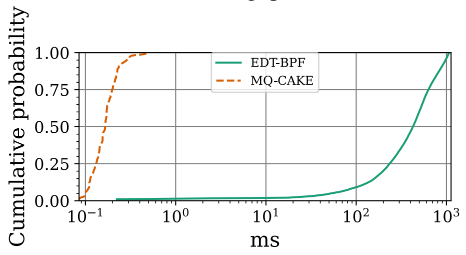


Figure 6: Achieved packet rate based on the available transmission queues for varying *synctimes*. The traffic consists of 4 flows containing only 64 byte packets. In case of a *synctime* of $0us$, the rate estimation is done for every packet.

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. This buffering increases latencies and may lead to 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 synchronization time. Figure 5 outlines the induced queue lengths as well as the corresponding induced queueing delays at three different global rate limits. Synchronization times beyond $100\mu s$ 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 also clearly 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 $50\mu s$ in the



(a) Throughput



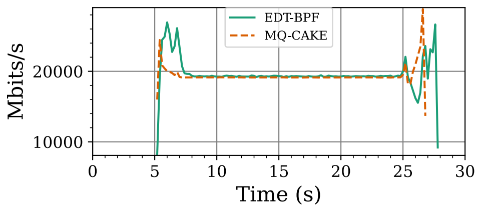
(b) Ping

Figure 7: Flent tcp_nup test with 1024 TCP streams, a configured rate limit of 20 Gbps, and a $2s$ drop horizon for EDT-BPF

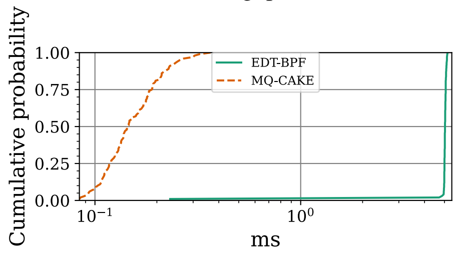
conducted experiments), the overhead of the synchronization loop increases, lowering the achieved throughput. Figure 6 shows the relation between the achieved rate and the number of transmission queues for different *synctimes*. This plot clearly shows that when the *synctime* is too low, the achieved 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 metrics.

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 [5] and the TCP Cubic [19] algorithm. Flent's tcp_nup test is executed using 1024 TCP upload streams. Figure 7 compares the performance of *mq-cake* and EDT-BPF under a global rate limit of 20 Gbps.



(a) Throughput



(b) Ping

Figure 8: Flent tcp_nup test with 1024 TCP streams, a configured rate limit of 20 Gbps, and a $5ms$ drop horizon for EDT-BPF

mq-cake maintains stable rate enforcement, remaining slightly below the configured limit, whereas EDT-BPF initially exceeds the rate limit before gradually reducing throughput in the latter half of the test (Figure 7a). Figure 7b shows the latencies measured during the test execution. *mq-cake* achieves $0.4ms$ latencies at the 99th percentile, a 2500x improvement as compared to EDT-BPF. The drop horizon of EDT-BPF directly corresponds with the expected latencies.

Figure 8 shows the same test execution as before but with a reduced drop horizon of $5ms$ for the EDT-BPF approach. Figure 8a reveals that lowering the drop horizon to $5ms$ not only stabilizes EDT-BPF's rate conformance but also reduces the tail latencies to $5ms$ (Figure 8b). However, reducing the drop horizon only works effectively, if the RTT's of the TCP flows have similar values as the drop horizon, thus high RTT flows would suffer from such a low drop horizon. 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.

Limitations

Over the course of the evaluation, we showed that *mq-cake* scales excellent with increasing number of hardware queues as well as reducing tail latencies. However, we identified that the current approach has a reduced accuracy when network traffic is suboptimal distributed across the *mq-cake* instances. So far, our experimental setup ensures that the qdisc layer of the Linux kernel is saturated with packets. However, under real-world conditions, this might not always be the case, as not all flows are sent at full speed or evenly distributed across transmission queues. 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.

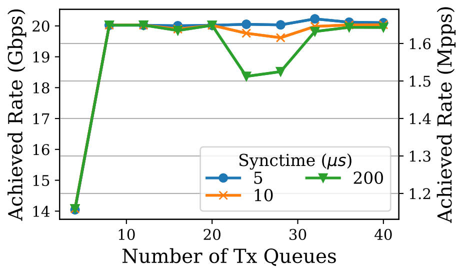


Figure 9: Achieved throughput in relation to the number of available hardware transmission queues with different *synctimes* for flows with full MTU-sized packets, where the rate limit is set to 20 Gbps and the number of receive queues is held at 40

For example, consider a case where the global rate limit is set to 10 Gbps and there is an incoming traffic of 10 Gbps. In this example, 80% of the incoming traffic is enqueued in Qdisc A and the remaining 20% of the traffic is steered to Qdisc B. In this case, both Qdiscs will estimate two active queues and lower their rate limit to 5 Gbps each. However, since Qdisc B can only forward 2 Gbps and Qdisc A is capped at 5 Gbps, the resulting throughput is only 7 Gbps. Figure 9 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 receive queues no longer distribute traffic equally across the transmission queues, which leads to the imbalanced scenario described above. For example, when there are 24 transmission queues, 16 transmission queues receive double the amount of packets 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. At the same time, the estimated rate for each transmission queue also decreases, leading to less unused bandwidth. These imbalances in multi-queue networking environments are well known in the literature [10, 21, 22].

Discussion and Future Work

The presented experiments reveal that *mq-cake* is able to shape traffic up to 25 Gbps while achieving high accuracy with a deviation around 0.25%. Further, *mq-cake* increases throughput with a greater number of hardware queues, achieving 14x higher packet rates as compared to single-queue CAKE and HTB. *mq-cake* also improves tail-latencies while being as accurate as the EDT-BPF approach.

The analysis of the *synctime* shows that setting its value too low increases the CPU overhead. As a lower bound, the synchronization time should be higher than the time it takes to complete one scan over all qdiscs. On the other hand, higher *synctimes* lead to less CPU load but also increase the time it takes *mq-cake* to converge. To balance between CPU load and accuracy, the experiments show that a *synctime* value between 100–200μs is ideal.

Looking ahead, we aim to further explore solutions for addressing load imbalances between *mq-cake* instances as well as approaches to mitigate overshooting above the configured rate limit during switching events. Concerning imbalances, in initial tests, we could already observe that it is feasible to add a rebalance mechanism to *mq-cake*, which can improve local rate limit estimations. In addition, we seek to investigate automated approaches to adjust and configure the *synctime* interval and other internal configurations such as for example the *memlimit*. In our experiments, we observed that reducing the *memlimit* had positive impacts on accuracy. 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.

Upstreaming the code

The results presented above are based on an implementation that localises all changes to the sch_cake qdisc itself. This involves the *mq-cake* initialisation code walking the qdisc tree to find its own siblings to initialise the cross-qdisc synchronisation data structure.

While this works well to evaluate the concept, this approach is not appropriate for inclusion into the mainline Linux kernel (*upstreaming*). So when proposing these changes for inclusion into the kernel, we aim to include a proposal for a better API for this shared state.

Specifically, we propose introducing a generalised notion of shared qdisc state that will be managed by the parent qdisc (in this case the *mq* qdisc). This shared state will be allocated and freed by the parent *mq* instance, and is passed to the sub-qdisc when it is attached to the parent. The shared state is keyed to a qdisc module owner, which means there will be one instance of shared state across all identical qdiscs attached to each *mq* instance.

To use this state, a qdisc module only needs define how much memory is needed by its shared data structure (by setting a *shared_size* parameter in its qdisc operations parameters), and define a function to receive the assignment from the parent, and (optionally) an initialisation function that is called the first time the shared state is allocated for a given parent instance.

The assignment function will be called whenever the qdisc is attached to a parent that manages shared state, and will contain a pointer to an object of the size defined by the qdisc module. The child qdisc can assign this pointer to its internal state and use it while running for any cross-qdisc operations. When the sub-qdisc is detached, it will be notified of removal of the shared state, so it can detach itself.

It is up to the qdisc module itself to manage concurrency across multiple instances accessing the shared state while running, which is no different from the implementation we have used for the evaluations in this paper. This also means that using this shared state API does not change the functioning of the multiqueue synchronisation algorithm itself, only the initialisation code used to setup the data structures it uses to do its work.

Conclusion

In this work, we present and evaluate a scalable, lock-less synchronization mechanism which allows for the correct enforcement of a global rate limit when scaling to multiple hardware queues. We integrated this synchronization mechanism into the CAKE queueing discipline, thus enabling running CAKE in combination with the MQ qdisc. We showed 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 2500x as compared to EDT-BPF. We believe that the proposed synchronization mechanism is a promising approach to effectively share state across queueing disciplines. As our next steps, we plan to investigate mitigation strategies to address imbalances and minimize overshoot above the rate limit during switching events, enhancing overall accuracy. Additionally, we aim to evaluate our approach using even faster network cards, assess its performance under real-world traffic conditions, and deepen our understanding of parameter configurations.

Acknowledgement

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

References

[1] tc-cake(8) — linux manual page. `https://man7.org/linux/man-pages/man8/tc-cake.8.html`.

[2] 2023. Cfp: Bandwidth manager with fq_codel. `https://github.com/cilium/cilium/issues/29083`.

[3] Emmerich, P.; Gallenmüller, S.; Raumer, D.; Wohlfart, F.; and Carle, G. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, 275–287. New York, NY, USA: Association for Computing Machinery.

[4] Fomichev, S.; Dumazet, E.; de Bruijn, W.; Dumitrescu, V.; Sommerfeld, B.; and Oskolkov, P. 2020. Replacing htb with edt and bpf.

[5] Høiland-Jørgensen, T.; Grazia, C. A.; Hurtig, P.; and Brunstrom, A. 2017. Flent: The flexible network tester. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2017, 120–125. New York, NY, USA: Association for Computing Machinery.

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

[7] Høiland-Jørgensen, T. 2023. The big fifo in the cloud. `https://blog.tohojo.dk/2023/12/the-big-fifo-in-the-cloud.html`.

[8] Linux iommu support. `https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt`.

[9] irqbalance. `https://linux.die.net/man/1/irqbalance`.

[10] Jang, K.; Sherry, J.; Ballani, H.; and Moncaster, T. 2015. Silo: Predictable message latency in the cloud. *SIGCOMM Comput. Commun. Rev.* 45(4):435–448.

[11] Jeyakumar, V.; Alizadeh, M.; Mazières, D.; Prabhakar, B.; Greenberg, A.; and Kim, C. 2013. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 297–311. Lombard, IL: USENIX Association.

[12] Kumar, A.; Jain, S.; Naik, U.; Raghuraman, A.; Kasinadhuni, N.; Zermeno, E. C.; Gunn, C. S.; Ai, J.; Carlin, B.; Amarandei-Stavila, M.; Robin, M.; Siganporia, A.; Stuart, S.; and Vahdat, A. 2015. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. *SIGCOMM Comput. Commun. Rev.* 45(4):1–14.

[13] Lévai, T.; Pongrácz, G.; Megyesi, P.; Vörös, P.; Laki, S.; Németh, F.; and Rétvári, G. 2018. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications* 36(12):2621–2630.

[14] McHardy, P. 2009. net_sched 00/07: classful multiqueue dummy scheduler. `https://lwn.net/Articles/351021/`.

[15] iproute2-6.5. `https://github.com/mq-cake/iproute2/tree/tc-mq-cake`.

[16] network namespaces. `https://man7.org/linux/man-pages/man7/network_namespaces.7.html`.

[17] Nichols, K., and Jacobson, V. 2012. Controlling queue delay. *Communications of the ACM* 55(7):42–50.

[18] Segmentation offloads. `https://docs.kernel.org/networking/segmentation-offloads.html`.

[19] Rhee, I.; Xu, L.; Ha, S.; Zimmermann, A.; Eggert, L.; and Scheffenegger, R. 2018. CUBIC for Fast Long-Distance Networks. RFC 8312.

[20] Saeed, A.; Dukkipati, N.; Valancius, V.; The Lam, V.; Contavalli, C.; and Vahdat, A. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 404–417. New York, NY, USA: Association for Computing Machinery.

[21] Stephens, B.; Akella, A.; and Swift, M. 2019. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 33–46. Boston, MA: USENIX Association.

[22] Stephens, B.; Singhvi, A.; Akella, A.; and Swift, M. 2017. Titan: Fair packet scheduling for commodity multiqueue NICs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 431–444. Santa Clara, CA: USENIX Association.