QVISOR: Virtualizing Packet Scheduling Policies

QVISOR: Virtualizing Packet Scheduling Policies

Albert Gran Alcoz, Laurent Vanbever

ABSTRACT

The concept of programmable packet scheduling has been recently introduced, enabling the programming of scheduling algorithms into existing data planes without requiring new hardware designs. Notably, several programmable schedulers have been proposed, which are capable of running directly on *existing commodity switches*. Unfortunately, though, their focus has been limited to single-tenant traffic scheduling: i.e., scheduling all incoming traffic following one single scheduling policy (e.g., pFabric to minimize flow completion times).

In this paper, we emphasize the fact that today's networks are heterogeneous: they are shared by multiple tenants, who run applications with different performance requirements. As such, we introduce a new research challenge: how can we extend scheduling programmability to multi-tenant policies?

We envision QVISOR, a *scheduling hypervisor* that enables *multi-tenant programmable scheduling* on *existing switches*. With QVISOR, tenants program the scheduling policies for their traffic flows; operators define how tenants should share the available resources; and QVISOR does the rest: deploying the scheduling policies into the underlying hardware.

1 INTRODUCTION

Packet scheduling has been an active area of research since the early days of the Internet. However, despite the numerous scheduling algorithms proposed over the years, only a few of them have made it into production. The reason is that deploying a new scheduling algorithm requires dedicated hardware support, yet developing new switch ASICs takes years and costs a lot of money (up to 200 million USD [2]).

Recently, programmable scheduling has been proposed, allowing operators to specify (new) scheduling policies on high-level abstractions that can be deployed to programmable hardware [31]. Significantly, a set of the programmable schedulers proposed do not require new hardware, and can directly run on *existing commodity switches* [3, 12, 26, 27, 39, 40]. They do so by smartly engineering the resources of programmable data planes to tag packets with ranks (i.e., priorities) based on a given policy, and by using the available scheduling resources to drop or prioritize packets following their ranks.

While promising, these works only cover the case of *single-tenant scheduling*, where *all the traffic* needs to be scheduled following *one single scheduling policy*. For example, they may schedule traffic following the Shortest-Remaining Processing Time policy to minimize FCTs [4, 5, 25]; *or* the FIFO+ policy to minimize tail latency [8, 21]; *or* a Fair Queuing scheme to enforce fairness across pre-defined traffic classes [10, 14, 20].

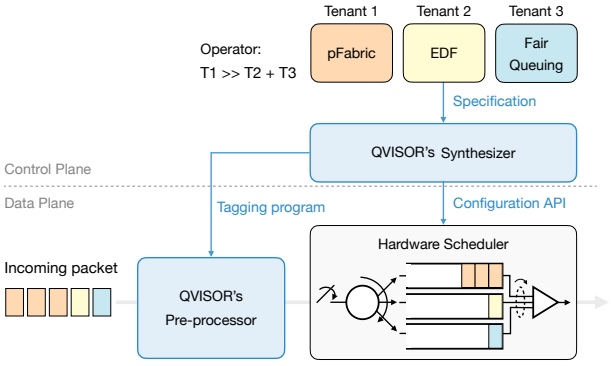


Figure 1: QVISOR's high-level architecture.

In practice, however, most networks today (e.g., cloud, data-center networks, and wide area networks) are shared by *multiple* tenants running various applications [19]. Each of these tenants may need to schedule their traffic using different policies in order to achieve their performance goals.

As such, in this paper, we introduce a new challenge: can we *simultaneously* deploy *multiple* scheduling algorithms on the scheduling resources of existing commodity switches?

Enabling multi-tenant scheduling on commodity switches requires solving two main challenges: (i) providing tenants a substrate where they can *specify* their scheduling policies, and (ii) finding mechanisms to *merge* and *deploy* the specified policies on top of the underlying hardware resources.

In other domains, this problem has long been solved thanks to *virtualization*, which abstracts the hardware resources and allows multiple tenants to coexist on the same infrastructure. The key enabler is the *hypervisor*, an interface between the tenants and the hardware, that deploys the applications of the tenants and orchestrates the hardware resources across them. Similarly, to enable multi-tenant scheduling on commodity switches, we need, essentially, a scheduling hypervisor.

A scheduling hypervisor Existing hypervisors typically virtualize compute resources at the end-hosts (CPU, memory, and I/O), allowing multiple tenants to access them while running their applications on virtual machines. Tenants only need to worry about programming their application, without having to mind about what other tenants do, nor about the details of the underlying infrastructure. Operators just have to specify tenants' access to resources (e.g., offering isolation and/or certain guarantees) and how resource conflicts should be resolved (e.g., by treating some tenants preferentially).

A scheduling hypervisor should follow the same intuition, but virtualizing the scheduling resources of the switch (i.e., the buffer and compute resources to execute the scheduling policies). Tenants should specify the scheduling policies to be used for their traffic on a high-level abstraction, as if they were to run in isolation on dedicated hardware. The operator should define how the scheduling resources should be shared across tenants (e.g., prioritizing traffic from certain tenants). The hypervisor should take care of the rest: combining and deploying the specified policies into the hardware resources.

QVISOR We envision QVISOR, a scheduling hypervisor to virtualize the scheduling resources of commodity switches, allowing them to be shared by multiple tenants (see Fig. 1). QVISOR takes as input a specification on how tenants wish to schedule their traffic, along with a high-level policy by the operator on how the scheduling resources should be shared. QVISOR then comes up with a joint scheduling strategy that follows the per-tenant policies while satisfying the operator's constraints, and deploys it into the underlying hardware.

Key challenges Realizing the vision for QVISOR, requires solving two challenges. First, we need to find flexible and easy ways for tenants and operators to *specify* their policies. Second, we need a means to *merge* and *deploy* these policies into the available infrastructure, while being able to *reason* about their performance. The solutions for both challenges have to be co-designed: we can not offer tenants a more-expressive abstraction than what can be later deployed, since it would lead to non-compilable applications, nor a more-limiting one, since it would lead to inefficient resource usage.

Insights We introduce a preliminary design of QVISOR that builds upon the following insights:

- Tenants have the illusion that their traffic is scheduled by a PIFO queue, which they can program using ranks.
- Operators define their policy with a composition language that enables resource sharing and prioritization.
- With these definitions, the problem is reduced to generating a single *joint* scheduling function, that combines the scheduling policies of the tenants and the operator.
- The specification is a 2-layer scheduler where the leafs (resp. root) are the intra- (resp. inter-) tenant policies.
- Multi-tenant policies have higher expressivity than single-tenant ones, but this only affects the worst case. *In practice*, workloads are not always active and do not always overlap, allowing us to multiplex the scheduling resources over time. When workloads do overlap, we resort to the high-level policy of the operator.
- Once we have a synthesized *joint* scheduling function, we can seamlessly deploy it to commodity schedulers.
- Overall, we essentially "trick" single-tenant schedulers to work as multi-tenant programmable schedulers.

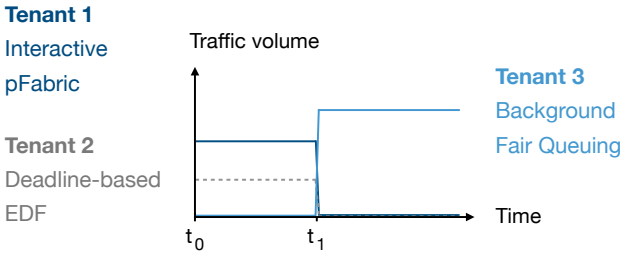


Figure 2: Example of a data-center workload.

2 MOTIVATION

Put yourself in the shoes of an operator managing a network with three tenants, T_1, T_2, and T_3 (see Fig. 2). Each tenant runs a different application. Your task is to handle the congestion in a switch, where the workloads of the three tenants coexist. The switch supports a PIFO queue, which you can configure to distribute the available bandwidth across the tenants.

All tenants have specified the scheduling policies that they wish to use for their traffic. T_1 runs an interactive application, sensitive to delay. Thus, T_1 would like to use pFabric's scheduling policy, which prioritizes flows with shorter remaining time, to minimize FCTs [4]. T_2 runs deadline-constrained flows, where meeting deadlines is crucial but delay sensitivity is low. Therefore, T_2 picks the earliest-deadline-first (EDF) algorithm to maximize meeting deadlines [9]. T_3 runs background applications and opts for a Fair Queuing scheme [10].

Based on the tenants' contracts, you know that tenants T_1 and T_2 should share the resources fairly, and should have priority over T_3. Your task is challenging for various reasons:

Problem 1: Scheduling policies clash with each other Within $t_0 < t < t_1$, traffic from tenant T_1 should be scheduled using pFabric, and traffic from tenant T_2, using EDF. How can we achieve this behavior on a conventional scheduler? Both scheduling policies have different objectives, and prioritize packets based on different metrics: EDF prioritizes packets based on the deadlines of their flows, and pFabric does it based on the remaining flow size [4, 9]. As such, if we naively execute the two scheduling policies simultaneously, they clash. Indeed, most packets from the deadline-constrained flows end up taking the link resources, since the priorities defined by the EDF scheduling policy are higher than the ones set by pFabric [39]. Thus, to reason about how to combine policies together, we need a way to compare them fairly.

Idea 1: Homogenize scheduling policies We can "normalize" and "quantize" policies into a common scale and granularity. This constrains their behavior and helps us reason about their interaction. Once polices are homogenized, we can compare them fairly, and think about how to merge them into a joint scheduling policy.

Problem 2: The way policies clash changes over time Even if we have a means to homogenize scheduling policies, their behavior is not constant: it depends on traffic, and traffic always changes. Traffic shifts become more drastic as tenants enter or leave the network. For example, at t_1, tenants T_1 and T_2 become inactive and tenant T_3 starts transmitting. T_3 has different performance requirements than T_1 and T_2 and, thus, requires a different scheduling strategy. At the same time, traffic from T_3 should be scheduled with the lowest priority. How can we seamlessly *switch* between scheduling policies?

Idea 2: Anticipate traffic shifts or react upon them We can develop analysis techniques to evaluate how different scheduling policies may work together, either theoretically, offline (e.g., based on worst-case analysis from the given specification) or online at runtime (e.g., based on the latest packets received). We can use the results to anticipate the scheduler to potential traffic shifts.

If we have an upfront specification of the policies of all the tenants, we can develop static analysis techniques to reason about the worst-case scenario for the combined workloads. Then, we can smartly design combined scheduling functions that enforce the desired behaviors even in the worst conditions. For example, if tenant T_3 should be scheduled with lower priority than T_1 and T_2, we can *shift* all the priorities from the T_3's scheduling policy such that, even in the worst case, it does not impact the performance of the other tenants.

If we do not have the specification of all tenants in advance, we can design reaction methods to adapt the scheduling policy at runtime, as tenants enter or leave the network. Similarly to how we deploy forwarding rules when a packet from a new flow arrives to an SDN switch, we can adapt the switch's scheduling policy under certain events. For example, an event-driven controller could synthesize a new scheduling policy after the first packets of a new workload arrived, and deploy it into the data plane. While this may come with challenges, such as emptying the buffers (e.g., if an incoming tenant has priority), or resetting the state of stateful scheduling functions, we believe that recently-proposed runtime programmable data planes can help lighting the path [38].

These analysis techniques would also be valuable to prevent adversarial workloads from potentially malicious tenants. For example, they could help us develop monitoring techniques to identify such adversarial workloads in the network and automatically stop them in case they ever occurred.

Problem 3: We don't have a standard scheduler PIFO queues provide us with a comfortable abstraction that eases the solution to the different challenges. Indeed, PIFO queues schedule packets with the guarantee that high-priority packets will always be scheduled before low-priority ones. This allows us to reduce the problem of designing a scheduling hypervisor into the one of analyzing how prioritization functions interact with each other and how they can be combined effectively. Existing switches today, however, do not support PIFO queues. Instead, they run, at most, PIFO approximations on top of strict-priority queues or FIFO queues [3, 26, 27, 40]. These PIFO approximations support different operations, and they offer various types of performance guarantees. How can we deploy multi-tenant policies on top of these schedulers?

Idea 3: Develop standard APIs or build synthesizers We can build standard APIs that allow the deployment of combined scheduling policies into existing schedulers. Alternatively, we can leverage program synthesis to transform high-level scheduling specifications down to the available hardware resources of existing switches.

To run on top of an existing scheduler, a scheduling hypervisor would need a set of APIs to interact with the various configuration parameters of the scheduler (e.g., to dedicate a set of priority queues to a certain tenant). As such, we would need to abstract the operations that the scheduler can support, and provide them to the hypervisor as a design space. With them, the hypervisor would be able to come up with a possible configuration, that satisfied the specification and reason about its guarantees. Further, given the set of operations supported by the scheduler, the hypervisor could leverage program synthesis to come up with a scheduling policy that fit into the available resource constraints, even if it could only satisfy a part of the input specification [13].

3 QVISOR OVERVIEW

We present a preliminary design for QVISOR, that defines its architecture and grounds the foundation for future research.

At the high-level, we envision QVISOR to work as follows (cf. Fig. 1). First, every tenant must specify the scheduling algorithms that should schedule the traffic within their flows. Second, the network operator must define a policy on how the scheduling resources should be distributed across tenants. Based on these inputs, QVISOR should: (i) synthesize a joint scheduling policy that follows the scheduling algorithms specified by the tenants, while respecting the constraints given by the operator; and (ii) deploy this policy into the underlying hardware to be used to forward incoming packets.

Accordingly, we propose a design composed of two parts: a *synthesizer* that runs in the control plane and generates the joint scheduling policy based on the inputs, and a *pre-processor* that runs in the data plane and "prepares" packets such that they can be scheduled with the synthesized policy.

In the following, we describe QVISOR's inputs (§3.1), synthesizer (§3.2) and pre-processor (§3.3). We also discuss how to deploy QVISOR on existing hardware schedulers (§3.4).

3.1 QVISOR inputs

Per-tenant scheduling policies Tenants define the scheduling policy that they want to use to schedule their traffic as a tuple composed of a traffic subset and a scheduling algorithm. For example, tenant $T_1 = \{\mathcal{P}_1, pFabric\}$ defines a set of packets $\mathcal{P}_1$, which should be scheduled with the *pFabric* algorithm. Another tenant, $T_2 = \{\mathcal{P}_2, STFQ\}$, comprises the set of packets $\mathcal{P}_2$ which have to be scheduled using the *STFQ* algorithm. Note that a *tenant* refers to a traffic segment (e.g., from a given application), not necessarily a physical tenant.

Packet labels For QVISOR to process incoming packets, it requires tenants to identify their packets with two labels: the *tenant identifier*, and the *packet rank*. Intuitively, the tenant identifier allows QVISOR to decide how to schedule the packet with respect to packets from other tenants, and the *packet rank* describes how to schedule the packet with respect to other packets from the same tenant. Packet ranks define the priority with which packets should be scheduled based on the rank function picked by the tenant. Ranks can be computed at the end-host or at the same switch, but always have to be specified before reaching QVISOR's pre-processor.

Operator's specification The operator defines a high-level inter-tenant policy using a simple string with the tenant identifiers, separated by three possible operators: {>>}, {>}, and {+}. The first operator indicates that the preceding tenant(s) should have strictly-higher priority than the following one(s), mandating isolation. The second, states that the preceding tenant(s) should be preferentially treated with respect to the following tenant(s). In this case, the priority is applied in a best-effort manner. The third operator indicates that preceding and following tenant(s), should share the resources. For example, the specification $\{T_1 >> T_2 > T_3 + T_4 >> T_5\}$ indicates that tenant T_1 should have strictly-higher priority than tenants T_2, T_3, and T_4, which in turn should have strictly-higher priority than T_5. It also defines that tenant T_2 should have higher priority, whenever possible, than T_3 and T_4, and that tenants T_3 and T_4 should share their resources.

3.2 QVISOR synthesizer

Given the input policy specifications of the tenants and the operator, the goal of the synthesizer is to generate a joint scheduling function that combines the algorithms of the different tenants, such that they can be simultaneously used, while satisfying the high-level requirements of the operator.

We express the joint scheduling function as a set of rank transformation functions to be applied to the ranks of the incoming packets. These functions are then deployed into the pre-processor and applied at line rate to incoming packets. QVISOR's synthesizer supports two types of transformation functions: rank-shift and rank-normalization operations.

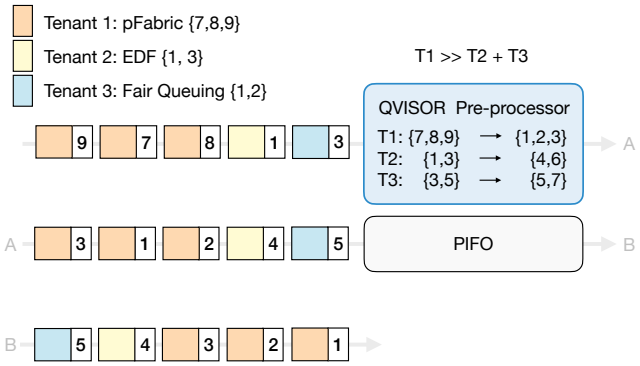


Figure 3: QVISOR's transformations.

Rank shift It allows the prioritization of traffic from certain tenants over others. For instance, to prioritize traffic from T_1 over T_2, we can *shift* the ranks of all packets of T_1 such that they have lower ranks than all packets from T_2. Alternatively, we can *shift* all ranks of T_2 such that they are higher than the ones of T_1. If the rank distributions are bounded and known in advance, we can implement most priority operations by just applying shifts to the ranks of the different tenants.

Rank normalization As we have seen in §2, naively scheduling packets with various rank functions simultaneously on the same scheduler can be detrimental. The normalization function consists in bounding the ranges of a given rank function, and quantizing its ranks into discrete levels such that they can be fairly compared with the (normalized) ranks of other tenants. With this function, different tenants can be scheduled simultaneously with a higher degree of fairness and without loosing their intra-tenant scheduling behavior.

3.3 QVISOR pre-processor

For each incoming packet, QVISOR's pre-processor parses the packet headers and extracts the *tenant identifier* and the *packet rank*. It uses these tags to query the transformation functions that should be applied to the packet, based on the results of the synthesizer. It executes them, updates the rank, and forwards the packet to the hardware scheduler.

Example Fig. 3 shows QVISOR's pre-processor handling a sequence of packets from tenants T_1, T_2, and T_3, ranked using pFabric, EDF, and FQ algorithms, respectively. The operator has specified that traffic from T_1 should have higher priority than traffic from T_2 and T_3, which should share the resources. Given these inputs, the synthesizer has generated three transformation functions: packets from T_1 carrying ranks $\{7, 8, 9\}$, have to be re-labeled with ranks $\{1, 2, 3\}$, respectively; packets from T_2 with ranks $\{1, 3\}$ have to be transformed into $\{4, 6\}$; and packets from T_3 with ranks $\{3, 5\}$, into $\{5, 7\}$. The pre-processor applies these transformations and forwards

the traffic to the scheduler (a PIFO queue), which sorts the packets based on ranks. As a result, the output sequence satisfies the input specifications: all packets from T_1 have the highest priority, and the traffic from tenants T_2 and T_3 share the resources evenly. At the same time, traffic within each tenant is scheduled in order of rank, following the specified ranking algorithm and achieving their desired performance.

3.4 QVISOR on existing schedulers

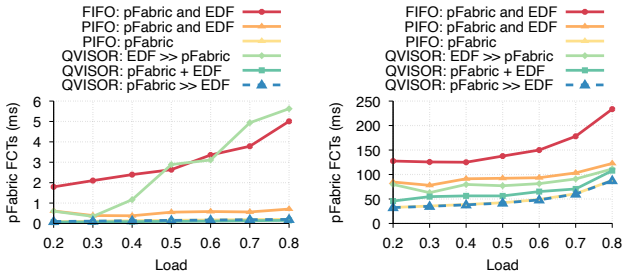
To run QVISOR on top of an existing scheduler, we have to consider two aspects. First, such schedulers consist of the baseline hardware (e.g., a set of FIFO queues), and some pre-processing function (e.g., to map packets to queues, or to decide which packets to admit). To virtualize them, we need to virtualize both, the hardware and the functions on top. Second, differently from PIFO queues, these schedulers do not always guarantee perfect packet sorting based on ranks. Thus, QVISOR may need additional configurations to fulfill the specification (e.g., dropping packets above a certain rank or dedicating queues to some tenants). As such, in order for QVISOR to run on existing schedulers, it should know what packet-processing operations they support and what guarantees they provide. With this information, QVISOR should be able to synthesize a set of operations to satisfy the specification while staying within the available resources.

For example, in Fig. 3 we need to prioritize traffic from T_1. In most existing schedulers [3, 27, 39], we can only guarantee such prioritization by allocating dedicated queues. This is, if we have a scheduler with five queues, we can map traffic from T_1 to the three highest-priority queues, and traffic from T_2 and T_3 to the two lowest-priority queues. Then, we need to run two mapping functions in parallel: one to map traffic from T_1 to the first three queues based on the pFabric policy, and another to map traffic from T_2 and T_3 to the last two queues, based on the EDF and FQ algorithms fairly combined.

4 PRELIMINARY EVALUATION

In this section, we show the potential of QVISOR and prove that it is actually possible to simultaneously run multiple scheduling algorithms on top of a single-tenant scheduler.

We implement a prof-of-concept version of QVISOR on Netbench [1], a packet-level simulator. We evaluate it when scheduling the traffic from two tenants on a data-center network. We use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches, and set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. The first tenant runs a data-mining workload that needs to be scheduled with the pFabric algorithm. The second tenant runs 100 flows that transmit at a constant bit-rate of 0.5Gbps between pairs of servers picked uniformly at random, which have to be scheduled following the EDF algorithm.



(a) (0,100KB): Mean FCTs **(b) [1MB,∞): Mean FCTs**

Figure 4: QVISOR's performance.

We measure the flow completion time of the pFabric traffic, under various loads, when the workloads are scheduled by: a FIFO queue, a PIFO queue, and QVISOR on top of a PIFO queue under three different policies: when the pFabric traffic is prioritized, when the EDF traffic is prioritized, and when both tenants share the resources. We also analyze the ideal case in which there is only pFabric traffic in the network.

Fig. 4 illustrates the resulting flow completion times for both, small and big flows. We can see how, FIFO and QVISOR with a policy that prioritizes EDF traffic, are the most detrimental cases for pFabric. This is expected: for the first, the FIFO scheduler can not prioritze traffic, and thus the pFabric policy becomes useless; and for the second, pFabric traffic gets blocked behind EDF traffic. We also see how naively executing the two scheduling policies on a PIFO queue is detrimental for pFabric, since most of the pFabric packets get deprioritized after the ones of EDF. Instead, when we use QVISOR with a policy that either prioritizes pFabric, or lets both tenants share the resources fairly, pFabric's traffic achieves a performance that is either ideal (i.e., equivalent to when pFabric traffic runs in isolation), or very close to ideal.

5 LOOKING FORWARD

The path towards scheduling virtualization is still plenty of open problems. In the following, we discuss a subset of them.

Increasing specification expressivity In our preliminary QVISOR design, tenants specify their scheduling algorithms using ranks, and operators define their policies using three basic operators (§3.1). We expect future research to explore novel QVISOR designs which can support more expressive specifications. For example, recent research has proposed more complex abstractions such as PIFO trees or Directed Acyclic Graphs (DAGs) [21–23, 30, 31, 33], which offer a higher degree of expressivity for both tenants and operators. With them, tenants can specify hierarchical and non-work-conserving scheduling algorithms, and operators can specify complex relations across tenants (e.g., multiple tiers). How to support these abstractions on QVISOR is an open question.

Compiling scheduling policies into hardware We also expect future research to focus on the interaction between QVISOR and existing schedulers. As discussed in §3.4, to work on existing infrastructures, QVISOR needs to be aware of the operations that they support. These operations have to be abstracted and provided to QVISOR as a domain-specific language, which it can then use to synthesize scheduling configurations. QVISOR also needs to know the guarantees that these operations offer, to be able to reason about whether the synthesized configurations can satisfy the specifications.

We could frame QVISOR's goal as a *compilation* problem where, given a high-level specification of the scheduling policies, and the design space of the operations supported by hardware, the objective is to come up with a set of operations that satisfies the policies. In a second step, we could take a *synthesis* approach, where QVISOR would not just fail if the desired policy could not be compiled, but would propose *partial specifications* implementable on the available resources. QVISOR would output the proposed configuration, together with the supported specifications and the offered guarantees.

Optimizing configurations at runtime We expect future designs of QVISOR to optimize at runtime both, the joint scheduling policy (e.g. computing transformation functions at line rate, based on the distribution of the latest packets), and the hardware-scheduler configurations (e.g., reallocating queues mapped to a tenant if the tenant is not transmitting).

Multi-objective scheduling algorithms In QVISOR, we have asked ourselves whether it is possible to run multiple scheduling algorithms on a single infrastructure. We have considered the case of having multiple tenants with different objectives. Another perspective would be to analyze whether we can achieve multiple objectives simultaneously *on the same traffic*. For example, Fair Queuing schemes enforce fairness, but also help in reducing FCTs, since they implicitly prioritize short flows. Multi-objective scheduling algorithms add another dimension to the QVISOR problem, offering new opportunities to combine traffic with similar requirements.

Synthesizing scheduling algorithms With the advent of programmable scheduling, we have more abstractions than ever to represent scheduling algorithms [21, 31] as well as algorithms for various objectives. Could we abstract and generalize this knowledge to create scheduling algorithms for arbitrary performance objectives? This is, given a workload and a performance goal (e.g., as a utility function or an SLA), could we synthesize the optimal scheduling algorithm?

Cross-device virtualization Recent works have managed to implement multi-tenant scheduling policies *at the end-hosts*, allowing the virtualization of the scheduling resources across tenants at the NICs [23, 33]. QVISOR enables multi-tenant scheduling *in the network* at commodity switches. As such, similarly to works in other domains [11, 35], we expect future research to propose mechanisms to orchestrate the scheduling virtualization from a *network-wide* perspective.

6 RELATED WORK

Packet scheduling Historically, most research on packet scheduling has focused on the design of algorithms with *a single performance objective*, such as enforcing fairness across traffic classes [8, 10, 14, 20, 28], minimizing FCTs [4, 5, 25]; or minimizing jitter [8, 21]. Recently, researchers explored the existence of a universal scheduler that could replicate any given algorithm [21, 32]. The lack of such a silver-bullet [32] triggered programmable scheduling [30] and the emergence of abstractions to make it possible [21, 23, 29, 31, 33]. Among them, a number of works focused on enabling it on commodity hardware [3, 12, 26, 27, 29, 40]. As a result, operators today can not satisfy *all* scheduling objectives at once, but can approximate most *individual* policies on existing devices.

In QVISOR, we bring up a new research question: is it possible to simultaneously implement *multiple* scheduling algorithms on a shared hardware scheduling infrastructure? We answer positively by proposing a scheduling hypervisor that navigates the space between programmable scheduling, and the ideal concept of a universal packet scheduler.

Virtualizing programmable networks Previous works have focused on virtualizing programmable networks, from their end-host counterparts to the in-network resources [6, 7, 15, 17, 18, 24, 34, 36–38, 41–43]. While most solutions target software switches, SmartNICs or NetFPGAs [17, 24, 34, 41], a few focus on hardware switches [16, 38, 41–43]. Among them, [16, 41, 42] optimize resource sharing at compile time by combining multiple applications into a single program, [38] enables runtime reprogramming of switch data planes, and [43] facilitates the dynamic sharing of switch resources across applications. Only [7, 17] address the virtualization of the scheduling infrastructure of programmable switches, but they do not virtualize programmable scheduling policies.

7 CONCLUSION

We introduce a vision for QVISOR, a scheduling hypervisor to virtualize the scheduling resources in commodity switches, enabling efficient resource sharing among multiple tenants. QVISOR acts as an intermediary between tenants and the underlying hardware. It allows tenants to program their own policies for their traffic, and operators to define a high-level policy on how resources should be shared. With these inputs, QVISOR creates a joint scheduling strategy that combines the tenant policies, while satisfying the operator's conditions, and deploys it to hardware. As a result, QVISOR brings multi-tenant programmable scheduling to conventional switches.

REFERENCES

[1] 2018. Netbench. http://github.com/ndal-eth/netbench.

[2] 2023. Network Programmability: The Road Ahead. https://www.youtube.com/watch?v=CtxfmES4T7E. (2023).

[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues . In *USENIX NSDI*. Santa Clara, CA, USA.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM*. Hong Kong.

[5] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX NSDI*. Oakland, CA, USA.

[6] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. 2016. Survey on Network Virtualization Hypervisors for Software Defined Networking. In *IEEE Communications Surveys and Tutorials*.

[7] Yan-Wei Chen, Chi-Yu Li, Chien-Chao Tseng, and Min-Zhi Hu. 2022. P4-TINS: P4-Driven Traffic Isolation for Network Slicing With Bandwidth Guarantee and Management. *IEEE TNSM*.

[8] David D. Clark, Scott Shenker, and Lixia Zhang. 1992. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *ACM SIGCOMM*. Baltimore, MD, USA.

[9] Fernando J. Corbató, Marjorie M.Daggett, and Robert C. Daley. 1962. An Experimental Time-sharing System. In *ACM AIEE-IRE*. New York.

[10] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queuing Algorithm. In *ACM SIGCOMM*.

[11] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. Virtual.

[12] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. 2022. Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing. In *USENIX NSDI*. Renton, WA, USA.

[13] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, and Pravein et al. Kannan. 2020. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*. Virtual.

[14] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *ACM SIGCOMM*. Palo Alto, CA, USA.

[15] Sol Han and Seokwon Jang et al. 2020. Virtualization in Programmable Data Plane: A Survey and Open Challenges. In *IEEE OJ-COMS*.

[16] David Hancock and Jacobus Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *ACM CoNEXT*. Irvine, CA.

[17] Hasanin Harkous, Chrysa Papagianni, Koen De Schepper, Michael Jarschel, Marinos Dimolianis, and Rastin Pries. 2021. Virtual Queues for P4: A Poor Man's Programmable Traffic Manager. In *IEEE TNSM*.

[18] Xin Jin, Jennifer Gossels, Jen Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for SDN. In *USENIX NSDI*. Oakland, CA.

[19] Sarah McClure, Zeke Medley, Deepak Bansal, Karthick Jayaraman, Ashok Narayanan, Jitendra Padhye, Sylvia Ratnasamy, Anees Shaikh, and Rishabh Tewari. 2023. Invisinets: Removing Networking from Cloud Networks. In *USENIX NSDI*. Boston, MA.

[20] Paul E McKenney. 1990. Stochastic Fair Queueing. In *IEEE INFOCOM*.

[21] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *USENIX NSDI*. Santa Clara, USA.

[22] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal Abstractions for Packet Scheduling. In *arXiv*.

[23] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *USENIX NSDI*. Boston, USA.

[24] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Azambuja. 2020. P4VBox: Enabling P4-Based Switch Virtualization. In *IEEE Communications Letters*.

[25] Linus E Schrage and Louis W Miller. 1966. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. In *INFORMS OR*.

[26] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX NSDI*. Renton, WA, USA.

[27] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *USENIX NSDI*. Santa Clara, CA, USA.

[28] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*. Cambridge, MA, USA.

[29] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *ACM SIGCOMM*. Beijing, China.

[30] Anirudh Sivaraman, Suvinay Subramanian, Anurag Agrawal, Sharad Chole, Shang-Tse Chuang, Tom Edsall, Mohammad Alizadeh, Sachin Katti, Nick McKeown, and Hari Balakrishnan. 2015. Towards Programmable Packet Scheduling. In *ACM HotNets*. Philadelphia, USA.

[31] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, S.T. Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*. Florianópolis, Brazil.

[32] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. 2013. No Silver Bullet: Extending SDN to the Data Plane. In *ACM HotNets*. College Park, MD, USA.

[33] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *USENIX NSDI*. Boston, USA.

[34] Radostin Stoyanov and Noa Zilberman. 2020. MTPSA: Multi-Tenant Programmable Switches. In *EuroP4*. Barcelona, Spain.

[35] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *USENIX NSDI*.

[36] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. 2022. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *USENIX NSDI*. Renton, WA, USA.

[37] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. 2020. Multitenancy for Fast and Programmable Networks in the Cloud. In *USENIX HotCloud*. Virtual.

[38] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime Programmable Switches. In *USENIX NSDI*. Renton, WA, USA.

[39] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. 2022. QCluster: Clustering Packets for Flow Scheduling. In *ACM WWW*. Virtual.

[40] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *SIGCOMM*.

[41] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *IEEE ICCCN*. Vancouver, Canada.

[42] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *ACM CoNEXT*. Heraklion, Greece.

[43] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. 2022. NetVRM: Virtual Register Memory for Programmable Networks. In *USENIX NSDI*. Renton, WA, USA.