Programmable Real-time Scheduling of Disaggregated Network Functions: A Theoretical Model

Tamás Lévai, Balázs Vass, Gábor Rétvári

***Abstract*—Novel telecommunication systems build on a cloudified architecture running softwarized network services as disaggregated virtual network functions (VNFs) on commercial off-the-shelf (COTS) hardware to improve costs and flexibility. Given the stringent processing deadlines of modern applications, these systems are critically dependent on a closed-loop control algorithm to orchestrate the execution of the disaggregated components. At the moment, however, the formal model for implementing such real-time control loops is mostly missing.**

In this paper, we introduce a new real-time VNF execution environment that runs entirely on COTS hardware. First, we define a comprehensive formal model that enables us to reason about packet processing delays across disaggregated VNF processing chains *analytically*. Then we integrate the model into a gradient-optimization control algorithm to provide optimal scheduling for real-time infocommunication services in a *programmable* way. We present experimental evidence that our model gives a proper delay estimation on a real software switch. We evaluate our control algorithm on multiple representative use cases using a software switch simulator. Our results show the algorithm drives the system to a real-time capable state in just a few control periods even in case of complex services.

***Index Terms*—dataflow graph, software switch, SDN, NFV**

I. Introduction

Current and upcoming telecom networks, such as 5G and O-RAN [1], rely on software-defined networks and virtual network functions (VNFs). These technologies enable the rapid and flexible development of network applications, bringing new real-time industrial applications (e.g. remote surgery), which seemed infeasible a few years ago, within reach (e.g. industrial robotic arm control [2]). There is a strong demand for these applications from both telecoms and manufacturing companies. A common feature of these new applications is that they impose stringent requirements on the network. For example, augmented reality (AR) applications require both 10 ms end-to-end delay and Gbits-scale bandwidth for media streaming [2]. Since this includes the media processing time on the endpoints, the data plane can use only a small fraction of the time budget. Likewise, industrial robotic arm motion control leaves a one-way delay between endpoints of 250-1000 µs [2]. Since this includes the processing time of the endpoints, the data plane has a fraction of this time frame. The 5G core network also imposes similar stringent requirements, where the end-to-end latency is 5-10 ms [3]. Although the latency limit is higher, there is a significant transmission delay (e.g., due to physical distribution) and processing time, which makes only a fraction of the time available for the software data plane. It is, therefore, important for the data plane that the transmission is fast and the traffic of critical applications is real-time.

Unfortunately, software VNFs running on commercial off-the-shelf (COTS) hardware usually cannot meet such firm latency requirements, which leads to unpredictable delay and throughput [4]. The main problem is that reasoning about performance in software is much more difficult than modeling hardware performance [5], [6]. A brute force solution is to simply allocate more CPU cores and hope for the best [7]. Given the firm delay and cost-efficiency requirements, however, this naïve approach is not sustainable. Another potential approach is to offload VNF processing to dedicated hardware (e.g., [8], [9], [10]). Requiring specialized hardware, however, limits flexibility and feature velocity. Recently there has been substantial work towards defining artificial intelligence and machine learning workflows to realize the real-time control loops [11], [12]; these solutions, however, do not make it possible to reason about performance *analytically* and still only exist as prototypes. In general, *viable frameworks for implementing real-time programmable control loops is still missing* [1], [2], [3], [13], [14], [15], [16]. Our main goal in this paper is to fill this gap. Our main contributions are a model for real-time VNF scheduling and a formal model-based closed-loop scheduling algorithm, which can guarantee the strict service-level objectives (SLOs) required to implement real-time control loops with a precision similar to hardware. The key of our design is a programmable real-time software switch running on COTS hardware, installed with an operating system with kernel-bypassing network stack (e.g., Intel DPDK [17]). Complex network functions and applications are implemented by manipulating the packet processing pipeline of the software switch. We observe that real-time software data processing is entirely contingent on the CPU scheduler. Consequently, we tackle two important problems: *i) resource allocation:* dynamically decompose the packet processing pipeline into scheduling units (tasks) and allocate each task to a CPU core, and *ii) optimal scheduling:* compute the optimal share of CPU

Tamás Lévai, Balázs Vass, and Gábor Rétvári are affiliated to the Budapest University of Technology and Economics (BME), and HUN-REN-BME Information Systems Research Group. Balázs Vass is also affiliated to Babeș-Bolyai University, Cluj Napoca, Romania.

Co-funded by the European Union

This work was partly supported by Project №135606 ANN_20, which has been implemented with support from the National Research, Development and Innovation Fund of Hungary. This study has received funding from the European Union's Horizon Europe research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101155116.

resources each task should receive to meet SLOs. Particular contributions are as follows.

Analytical Model: We give a model (§II and §III) that allows to *formally reason* about the rate and delay in disaggregated VNF service chain. The model is validated and fine-tuned in a real software switch. The model is extendable for multi-switch case to guarantee end-to-end performance guarantees for services spanning over multiple switches.

Scheduler Control: We introduce a *model-predictive controller* (§IV) that adjusts the software switch scheduler to meet any given SLOs at the millisec granularity. We present a gradient-optimization-based control algorithm to optimally distribute CPU fair-queuing scheduling weights to guarantee rate and delay SLOs of services. We further improve the robustness of the system by distributing tasks between CPU cores.

Numerical Evaluation: We validate our model with measurements executed on the Berkeley Extensible Software Switch (BESS) [18]. Using a custom simulation environment (§V), we demonstrate the effectiveness of our solutions in synthetic examples and realistic cellular network use cases (§VI). Our simulator is available for download at [19].

We close the paper by discussing the related work (§VII) and deriving the conclusions (§VIII).

II. Background, Challenges and System Model

Our main goal is to define a formal framework for implementing real-time control loops for disaggregated network functions. The key of our design is a programmable real-time software switch running on COTS hardware, which schedules the execution of the network functions and runs a fast packet processing pipeline as a communication substrate between the functions. Real-time execution is enforced by integrating the pipeline into a closed control loop that adjusts the scheduling of each function, so that the end-to-end processing chain meets the delay and jitter SLOs imposed by the operator. Real-time control in this context boils down to deciding *a)* how much CPU power is needed per function so that each processing chain receives exactly the requested end-to-end SLOs; *b)* how to allocate functions to CPUs and set scheduling weights so that no CPU is overburdened, and *c)* how to dynamically track changes in the input packet rate and/or delay SLOs.

Answering these questions is not trivial due to the sheer amount of system parameters. On hardware level, the execution is affected by the CPU and the memory models (e.g., L1-L3 caching, NUMA, memory contention, branch prediction, etc.) [20]. On top of this, the operating system adds additional unknowns: interrupts (e.g., NMI and IRQ), the CPU scheduler, Spectre/Meltdown mitigation, etc. Moreover, all of these unknown system parameters are hidden during execution, only the high-level parameters are observable.

We overcome this limitation in two steps. First we assume a static setting. We manually decompose the VNFs to tasks, where each task is defined as the smallest unit of execution our scheduler can run. Given this decomposition, we define an optimal control algorithm to compute weighted-fair-queuing (WFQ) scheduling weights so that all queues in the system are drained fast enough to keep latency below the delay-SLOs, and each module is executed enough times to process total of the rate-SLOs. The real-time scheduling control loop is depicted in Fig. 1. In the next step we adapt our optimal scheduler to a dynamic setting where resource allocation is optimized online, always updating the previous state of the system in response to the changes in input traffic rate, delay-SLOs, and/or rate-SLOs.

A. Dataflow Graphs and Scheduling

Software switches for running disaggregated network functions (VPP [21], Click [22] and FastClick [23], BESS [18], NetBricks [24], etc.) are usually designed as a *dataflow-graph runtime*. This model is much akin to TensorFlow [25] for machine learning or GStreamer [26] for multimedia.

In this model, **modules** (or *nodes*) represent packet processing functions (VNFs). At the most basic level, each module implements a single processing primitive, like parse/deparse, match/action, encap/decap, filter/queue, which can then be freely combined into a meaningful high-level pipeline. At a higher level, modules can each implement complex network functionality; for instance, a L3 router, a full 5G mobile gateway data plane, etc. Our framework is completely agnostic to the choice of modules (i.e., VNFs). Control-flow is represented by linking modules with **edges** into complex network function chains, with each edge connecting an outgate of an upstream module to an ingate of a downstream module. A packet batch placed by the upstream to the outgate will appear at the corresponding downstream ingate immediately. Modules and edges together form the **dataflow graph**.

Virtual network function chains are abstracted as **flows** in our model, where each flow is a path from the flow's ingress module to the egress module and service delay and rate SLOs are defined over this end-to-end path. Software switches implement multi-level *hierarchical scheduler* with the scheduling loops working in tandem. At the upper level, each CPU core runs a CPU scheduler that is responsible for picking the next schedulable unit (called a **task**) on the given CPU core to execute next. Two common strategies are round-robin (equal share) and **weighted-fair-queuing** (WFQ, weighted share) [27]. CPU schedulers may operate in different resource domains (e.g., CPU cycles, batch count, or packet rate). Each task forms a scheduling unit, comprising a connected rooted sub-graph of the dataflow graph with the root module being a schedulable module (e.g., a NIC RX/TX queue) that can be executed by the scheduler. At this lower level of execution, the task's input module drains the queue and automatically executes the rest of the task's pipeline in non-preemptible mode (run-to-completion).

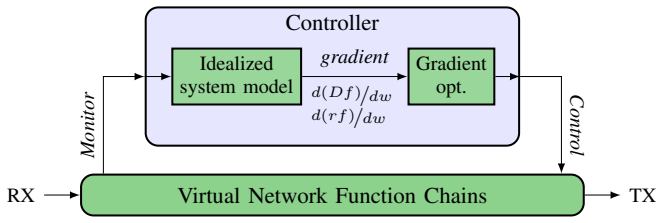


Figure 1. Real-time Control Loop.

B. System Model

We model the **dataflow graph** with a directed graph $G(V, E)$, where $v \in V$ represent the **modules** and dependencies between modules are modeled by arcs $(i, j) \in E$. For each module $v \in V$ its per-module processing cost c_v denotes the number of CPU cycles needed to process a single packet through module $v \in V$, measured in [cycle/pkt]. We assume for now that the module costs are given (we relax this later).

Each **flow** f is represented by a tuple of the following: directed path p_f through G, offered packet rate ρ_f [pkt/sec], rate-SLO (requested rate) R_f [pkt/sec], and delay-SLO D_f [sec]. The set of flows is denoted with F. Each **task** $G_t(V_t, E_t) : t \in T$ is a connected subgraph of G with a single input queue. Let $F_t \subseteq F$ be the set of flows traversing subgraph G_t, let $p_{t,f}$ be the path of a flow $f \in F_t$ through G_t, let $\pi_f = \{(t_1, t_2), (t_2, t_3), ...\}$ be the ordered sequence of task pairs traversed by f, and let s_f be the input task for f.

We assume there are S **workers** (i.e., CPU cores) available to run the packet processing pipeline. The set of tasks assigned to the i-th worker is denoted as $S_i = \{t_{i,1}, t_{i,2}, ...\}$; these tasks share the CPU time budget T_i of the worker (e.g., $T_i = 2.4 \cdot 10^9$ cycle/sec for a 2.4 GHz CPU). For a task t, let i_t denote the CPU that runs t. Furthermore, for each task t, w_t denotes its **task scheduling WFQ weight**, defining the share of CPU time that task t receives at its worker. For each worker, we normalize weights to 1: $\sum_{t \in S_i} w_t = 1, w_t \geq 0$.

Fig. 2 depicts a sample packet processing pipeline taken from [28] along with two different settings of scheduling weights, each yielding different packet processing performance. Namely, one of the settings is feasible while the other is not. The dataflow graph consists of 3 tasks, where per-module processing cost c_v for the modules in task_1 is 2 units and for the modules in task_2 is 1 unit, while modules in task_0 have negligible cost. The flows are defined as follows: flow_1 goes from NIC1 to NIC2 via task_0 and task_1, while flow_2 goes from NIC1 to NIC3 across task_0 and task_2. Here, flow_1 requests a rate-SLO of at least $R_1 = 1/3$ units and a delay-SLO of at most $D_1 = 5$ units, while flow_2 requests $R_2 = 1/5$ and $D_2 = 6$. There is a single worker. First, consider a scheduling regime where task_1 and task_2 receive equal CPU share (i.e., $w_1 = w_2 = 1/2$). Intuitively, flow_1 is restricted to a rate of $w_1/c_1 = 1/4 < R_1$ units which violates the requested rate-SLO, despite that the worst-case delay $c_2/w_2 + c_1/w_1 = 4 + 2 > D_1$ meets the delay-SLO. At the same time, flow_2 meets its SLOs and there is some slack remaining: it gets a packet rate of $1/2$ units and $4 + 1 = 5$ units delay. Second, consider an unequal CPU scheduling where we reallocate the resource slack from the second task to the first one. With CPU shares of $w_1 = 4/5$ and $w_2 = 1/5$, flow_1 receives $2/5 > R_1$ units of packet rate and $2/\frac{4}{5} + 2 < D_1$ units delay, and flow_2 gets $1/5 = R_2$ units rate and $5 + 1 = D_2$ units delay, both meeting the requested SLOs.

C. Problem Statement

This intuitive example shows that allocating the limited CPU resource share to executing each task has critical impact on end-to-end processing rates and delays. The main task here is to control the CPU share via the WFQ scheduling weights in order for each flow (i.e., network function chain [29]) to meet its requested rate-SLO and delay-SLO. We consider two problem formulations with increasing complexity in this context. Our first, simplified model assumes static resource allocation.

Table I
SYSTEM MODEL SYMBOLS

Notation	Meaning
$G(V, E)$	Dataflow graph with modules $v \in V$, arcs $(i, j) \in E$
c_v	CPU cycles needed to process a single packet through module $v \in V$ [cycle/pkt]
$f = (p_f, \rho_f, R_f, D_f)$	f: a flow, that is a tuple of the following: p_f: a directed path through G ρ_f: offered packet rate [pkt/sec] R_f: rate-SLO (requested rate) [pkt/sec] D_f: delay-SLO [sec]
F	set of flows
t, T	task; set of tasks
$G_t(V_t, E_t)$	graph of task t, that is a connected subgraph of G with a single input queue
F_t	the set of flows traversing subgraph G_t
$p_{t,f}$	the path of an $f \in F_t$ through G_t
$\pi_f = \{(t_1, t_2), (t_2, t_3), ...\}$	path of flow f, that is an ordered sequence of pairs o tasks traversed by flow f
s_f	the input task for flow f
$S_i = \{t_{i,1}, t_{i,2}, ...\}$, $i \in [1, S]$	set of tasks run worker i
T_i	time budget of worker i that is shared among the tasks
i_t	the CPU/worker that runs task t
w_t	the WFQ weight that defines the share of CPU time task t receives at its worker $i \in [1, S]$
Further definitions of the ideal system	
$r_{t,f}$	flow rate for flow f at task $t \in T$ [pkt/sec]
$\tau_{t,f}$	per-task processing cost for flow f at task t such that $f \in F_t$ [cycle/pkt]: CPU cycles to process one packet of f through t: $\tau_{t,f} = \sum_{v \in p_{t,f}} c_v$
θ_t	task processing time [cycle/pkt]: avg. number of CPU cycles a task $t \in S_i$ may need to process a single packet $\theta_t = \frac{\sum_{f \in F_t} r_{t,f} \tau_{t,f}}{\sum_{f \in F_t} r_{t,f}} = \frac{\sum_{f \in F_t} \left(r_{t,f} \sum_{v \in p_{t,f}} c_v \right)}{\sum_{f \in F_t} r_{t,f}}$
B	(constant) batch size [pkt/batch]
C	[batch], usually, $C = 1$
Q	queue size
$\hat{d}_{t,f}$	delay [sec] suffered by flow f at task t can estimated from the model, it equals $\max \left\{ \max_{t' \in S_i : t' \neq t} \left(CB \frac{\theta_{t'}}{T_{i_t}} \right), \frac{Q \theta_t}{T_{i_t}} \frac{1}{w_t} \right\} + CB \frac{\theta_t}{T_{i_t}}$

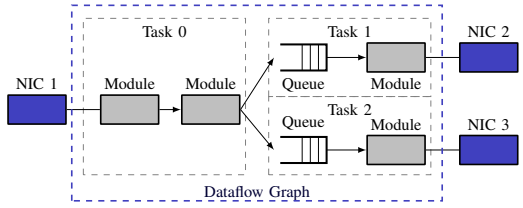
Problem 1 (Feasible scheduling with fixed resource allocation)**.** *Given a set of static task graphs G_t and a fixed allocation of tasks to workers S_i, compute scheduling weights w_t so that rate-SLO R_f and delay-SLO D_f is satisfied for each flow $f \in F$.*

In the second problem, resource allocation is adjustable online:

Problem 2 (Feasible scheduling with adaptive resource allocation)**.** *Compute a resource allocation (i.e., task graphs G_t and task allocations to workers S_i) and the related scheduling weights w_t, so that the rate-SLO R_f and delay-SLO D_f is satisfied for each flow $f \in F$.*

III. An Optimal Scheduling Algorithm

First, consider Problem 1. Given a static resource allocation we define an ideal scheduler for optimal module execution scheduling in a dataflow graph under delay and rate SLOs.



System setting: Flows: flow_1: NIC1 $\rightarrow$ NIC2, flow_2: NIC1 $\rightarrow$ NIC3.
Per-module processing cost c_v of modules in task_1 and task_2, equal to 2 and 1, respectively.
Modules in task_0 have negligible costs. There is a single worker.
Rate SLOs: flow_1 requests at least a $R_1 = 1/3$, $R_2 = 1/5$
Delay SLOs: flow_1: $D_1 = 5$, flow_2: $D_2 = 6$

Bad scheduling: equal CPU time on tasks 1 and 2 (i.e., $w_1 = w_2 = 1/2$) flow_1 is restricted to $w_1/c_1 = 1/4 < 1/3 = R_1$ unit of flow, and $c_2/w_2 + c_1/w_1 = 4 + 2 = 6 > 5 = D_1$ units of worst-case delay, however, flow_2 gets a feasible $1/2$ unit of flow and $4 + 1 = 5$ unit delay.

Good scheduling: $w_1 = 4/5$ and $w_2 = 1/5$: this way, flow_1 receives $2/5 > R_1$ $(1/3)$ unit of flow and $2/\frac{4}{5} + 2 = 4.5 < D_1$ (5) unit delay, and flow_2 gets $1/5 = R_2$ unit of flow and $5 + 1 = 6 = D_2$ unit delay.

Figure 2. Example Pipeline (see [28]).

A. Assumptions

First, we assume certain parameters are known, constant, and observable; the intention is to see whether the problem is at least theoretically solvable under very strong (and unrealistic) assumptions. Later on, we will relax the assumptions that prove detrimental in an implementation.

Assumption 1 (Unsplittable flows)**.** *Each flow takes only a single path through G.*

If a flow is split (e.g., for load-balancing) across multiple paths, we add each possible path as a separate flow with properly adjusted SLOs to the system.

Assumption 2 (Lossless modules)**.** *There is no internal packet loss and/or packet drops inside the modules of the pipeline.*

To be consistent with Assumption 2, dropped packets can be directed to a dedicated loss gate.

B. Delay Estimation

In order to fulfill the delay-SLOs, we need a formal delay estimation for each flow. First, we introduce some additional notation. For flow f at task $t \in T$, we denote the **flow rate** with $r_{t,f}$ [pkt/sec]. Again, we assume $r_{t,f}$ is known, and later we will relax this assumption. The **per-task processing cost** $\tau_{t,f}$ [cycle/pkt] for flow f at task t (where $f \in F_t$) is defined as the number of CPU cycles needed to process one packet of f through t. Observe that $\tau_{t,f} = \sum_{v \in p_{t,f}} c_v$. The **task processing time** θ_t [cycle/pkt] measures the average number of CPU cycles a task $t \in S_i$ needs to process a single packet: $\theta_t = \frac{\sum_{f \in F_t} r_{t,f} \tau_{t,f}}{\sum_{f \in F_t} r_{t,f}} = \frac{\sum_{f \in F_t} \left(r_{t,f} \sum_{v \in p_{t,f}} c_v \right)}{\sum_{f \in F_t} r_{t,f}}$. The **per-batch task processing time** $B\theta_t$ [cycle] denotes the average number of CPU cycles $t \in S_i$ needs to process a single batch of packets, where B [pkt] is the (constant) batch size. In addition, Q [packet] will denote the maximum queue size.

Using this notation, the **delay** for flow f is the sum of the delays at each task t traversed by f. The per-task delay of the flow is estimated by the sum of the *queuing delay* in the task's input queue plus the *processing delay* required to process a packet of f through the pipeline along path $p_{t,f}$. To compute these terms, we need the following lemma.

Lemma 1. *Given a stride-based WFQ scheduler [27], the time between two consecutive runs of a task t equals $\frac{B\theta_t}{T_{i_t} w_t}$ on average.*

The proof of Lemma 1 is relegated to Appendix A.

Let us compute the queuing delay first, initially assuming that each task t receives its fair share of CPU (i.e., proportional to w_t). In this case a packet may need to wait $\frac{Q}{B}$ turnaround times to be drained from the input queue. Observing that an average scheduling turn takes $\frac{B\theta_t}{T_{i_t} w_t}$ secs (by Lemma 1), the queuing delay for flow f at task $t : f \in F_t$ equals $\frac{Q}{B} \frac{B\theta_t}{T_{i_t}} \frac{1}{w_t} = \frac{Q\theta_t}{T_{i_t}} \frac{1}{w_t}$ [sec]. However, in certain cases a heavy-weight task t' may occupy the CPU for an extended time, starving the rest of the tasks. In such cases, the queuing delay of each remaining task $t \neq t'$ equals the amount of time t has to wait until t' finishes running (recall, there is no preemption inside tasks) and yields the CPU: $\max_{t' \in S_i : t' \neq t} (B \cdot \theta_{t'}/T_{i_t})$ [sec]. The queuing delay is then the maximum of the above two expressions.

Modeling the packet processing delay is simpler: for flow f the average packet processing delay at task t equals the cost of processing a batch of size B, through the pipeline of t each time t is scheduled: $B \frac{\theta_t}{T_{i_t}}$ [sec].

Hence, the **estimated total delay** of flow f at task t is: $d_{t,f} = \max \left\{ \max_{t' \in S_i : t' \neq t} \left(B \frac{\theta_{t'}}{T_{i_t}} \right), \frac{Q}{B} \frac{B\theta_t}{T_{i_t}} \frac{1}{w_t} \right\} + B \frac{\theta_t}{T_{i_t}}$. In the sequel, we use this delay estimation in the system model. We will simplify the delay estimate in §IV-A and justify the model empirically later in §VI-A.

C. Rate Estimation

In order to fulfill the rate SLO for each flow, each task must be allocated enough CPU time so that it can process all its offered load. Clearly, the **offered load** for task t is at most the sum of the offered packet rates of the flows that traverse the task: $\sum_{t \in F_t} r_{t,f}$. We also know that the amount of work to be done at task t for each packet of f is $\tau_{t,f} = \sum_{v \in p_{t,f}} c_v$. The total CPU share allocated to t is w_t, and this must be larger than or equal to the CPU time needed to process the total offered load of t, which, based on the former, yields the following constraint: $\frac{1}{T_{i_t}} \sum_{f \in F_t} r_{t,f} \tau_{t,f} \leq w_t$. This estimation is true only as long as there is no packet drop in the pipeline (recall Assumption 2), i.e., as long as flow conservation holds. For this, the producers (upstreams) of a task cannot generate more traffic than what the sinks (the downstream task) can process; formally: $r_{s,f} = r_{t,f} : f \in F, (s,t) \in \pi_f$ and $r_{s_f,f} = \rho_f$. Without loss of generality, we sum these constraints for each flow in the task to get a per-task constraint: $\sum_{f \in F_t} \sum_{s:(s,t) \in \pi_f} r_{s,f} + \sum_{f : s_f = t} \rho_f = \sum_{f \in F_t} r_{t,f}$, for all $t \in T$, where the term $\sum_{f : s_f = t} \rho_f$ accounts for the offered rate of the flows f that enter the pipeline at task t. This will be useful later when we satisfy the feasibility constraint by enabling *back-pressure* in BESS since BESS does not support per-flow back-pressure (like NFVnice [30]).

D. Feasible WFQ Scheduling Control with Fixed Resource Allocation

We are now in the position to formulate an optimization problem to answer Problem 1. Given dataflow graph G, flows F

with rate-SLOs R_f and delay-SLOs D_f, and resource allocation (G_t, S_i), and supposing that $c_v : v \in V$ are known with $\tau_{t,f} := \sum_{v \in p_{t,f}} c_v$, the task is to compute WFQ task weights w_t so that the constraints (1)–(5) below are satisfied.

$$\sum_{t:f\in F_t} \left(\max\left\{ \max_{t'\in S_i : t'\neq t} \left(B\frac{\theta_{t'}}{T_{i_t}} \right), \frac{Q\theta_t}{T_{i_t}} \frac{1}{w_t} \right\} + B\frac{\theta_t}{T_{i_t}} \right) \leq D_f, \ \ \forall f \in F \tag{1}$$

$$\sum_{f\in F_t} r_{t,f}\tau_{t,f} \leq w_t T_i, \ \ \forall t \in S_i, \forall i \in [1,S] \tag{2}$$

$$\sum_{f\in F_t} \sum_{s:(s,t)\in\pi_f} r_{s,f} + \sum_{f:s_f=t} \rho_f = \sum_{f\in F_t} r_{t,f}, \ \ \forall t \in T \tag{3}$$

$$\sum_{t\in S_i} w_t \leq 1, \ \ \forall i \in [1,S] \tag{4}$$

$$w_t \geq 0, r_{t,f} \geq \min\{\rho_f, R_f\}, \ \ \forall t \in T, f \in F. \tag{5}$$

IV. A Practical Real-time Scheduler

Unfortunately, the ideal system to solve Problem 1 is difficult to apply in practice. First, it assumes a static resource allocation. Second, it depends on the module costs c_v ($v \in V$) that are either not known or may vary with the workload, configuration of v, etc. Third, we cannot measure parameter $\tau_{t,f}$ and flow-rates $r_{t,f}$ directly from the running pipeline. Fourth, even if we could, constraint (1) is a non-convex function of $r_{t,f}$, which is hard to optimize. Fifth, the system tries to track the offered rate ρ_f even if $\rho_f > R_f$. To overcome these difficulties, below we simplify the ideal system step-by-step until we arrive to a convex formulation with all the remaining parameters easy to be measured from the running system. This simplified system will then lend itself readily to a fast online algorithm. As a next step, we will present an actual online optimization algorithm for this purpose; this tackles Problem 1. Finally, we turn to the fully-fledged problem formulation posed in Problem 2 and consider several practical dynamic resource allocation heuristics.

A. A Simplified Model

Back-pressure: An apparent problem is that constraint (3) must be enforced at a packet-by-packet basis, and this dynamics may be difficult to track from the scheduler. *Back-pressure* is an in-band method to enforce flow conservation [30], which automatically blocks upstream module execution when some downstream gets overflown (i.e., the input queue of a downstream module gets a backlog greater than a predefined watermark). Enabling back-pressure, we automatically satisfy (3) so we can remove this from the model, allowing us to treat the flow rates $r_f = \min_{t:f\in F_t} r_{t,f}$ as constant during control periods. This also has the added benefit that we no longer need to measure the input rate ρ_f from the running system, and now the task processing times $\theta_t = \texttt{const}$ can be directly measured from the pipeline.

Constant rate: Another problem is that, by (1), the queuing delay is non-convex in variables r_f. To overcome this problem, we will assume that the dynamics of the input traffic is such that the rate of flows can be considered constant inside a control period. Earlier work showed that this assumption is generally true when the control period is small enough (e.g., below 10-100 ms) [31], [32]. Now, r_f is no longer a variable to be optimized but a parameter to be measured from the running system. Then, since θ_t is now constant, non-convexity vanishes from (1). Below, we will use the shorthand notation $r_t := \sum_{f\in F_t} r_f$ to denote the total packet rate of task t.

Table II
Symbols for the simplifications

Notation	Meaning
r_f	realized rate of flow f, where $r_f = \min_{t:f\in F_t} r_{t,f}$
r_t	total packet rate at task t, where $r_t = \sum_{f\in F_t} r_{t,f}$
λ_t	parameter for the queue size $\lambda_t = 0$ and $\lambda_t = 1$ mean empty and full queues, respectively
α	parameter, the higher the more optimization for delay SLOs
L_i	objective function value for worker i

Dualization: A third issue is that without a precise measurement on $\tau_{t,f}$, we cannot decide if (2) is satisfied. To tackle this problem, we dualize (2) by moving it to the objective and using the queue size as dual. The idea is that if (2) is tight for a task t then the queue size (i.e., the dual λ_t) grows, so we increase the CPU share w_t. Note that λ_t is not the usual Lagrangian dual, but rather a parameter (queue size) we measure from the system. The simplified system at this point looks like the following:

$$\min \sum_{i\in[1,S]} \sum_{t\in S_i} \lambda_t \left(\frac{1}{T_i} \sum_{f\in F_t} r_f \tau_{t,f} - w_t \right) \tag{6}$$

$$\sum_{t:f\in F_t} \left(\max\left\{ \max_{t'\in S_i : t'\neq t} \left(B\frac{\theta_{t'}}{T_{i_t}} \right), \frac{Q\theta_t}{T_{i_t}} \frac{1}{w_t} \right\} + B\frac{\theta_t}{T_{i_t}} \right) \leq D_f, \ \ \forall f \in F \tag{7}$$

$$\sum_{t\in S_i} w_t \leq 1 \qquad i \in [1,S] \tag{8}$$

$$w_t \geq 0 \qquad t \in T. \tag{9}$$

Ignore processing delay: The delay at task t comprises the queuing delay plus the time needed to process the packet through t. In general, however, the queuing delay usually dominates the processing delays by 1-2 orders of magnitude [31]. Thus, it is safe to assume that no heavy task will dominate the queuing delay, and we can omit all the components from (7) except for queuing: $Q\frac{\theta_{t'}}{T_{i_t} w_t}$. This has the additional benefit that another difficult-to-measure parameter $\tau_{t,f}$ disappears from the model:

$$\min \sum_{i\in[1,S]} \sum_{t\in S_i} -\lambda_t w_t \tag{10}$$

$$\sum_{t:f\in F_t} \frac{Q\theta_t}{T_{i_t}} \frac{1}{w_t} \leq D_f \qquad f \in F \tag{11}$$

$$\sum_{t\in S_i} w_t \leq 1 \qquad i \in [1,S] \tag{12}$$

$$w_t \geq 0 \qquad t \in T \tag{13}$$

Enforce delay SLOs in the objective: Most interior point solvers will have trouble to account for the infeasibility possibly introduced by a violation of the delay constraint (11). To address this issue, we represent (11) with a linear penalty function: $P(f) = \alpha \max\left[0, \sum_{t:f\in F_t} \frac{\theta_t Q}{T_{i_t} w_t} - D_f\right]$. Here, $\alpha \geq 0$ is a tunable parameter: the higher α is, the

Algorithm 1: Projected Gradient Method for a Worker

Input: for all task $t \in S_i$, λ_t and θ_t given, and initial weight set as $w_t[1] = 1/|S_i|$

Find: $\arg\min\{L_i(w)|w \geq 0, \sum_{t=1}^{|S_i|} w_t = 1\}$

$k := 1$

for $t \in \{1, \ldots, |S_i|\}$ **do**

$-\nabla L_i(w_t[k]) := \lambda_t + \frac{1}{w_t[k]^2} \frac{1}{T_{i_t}} \cdot \alpha\theta_t \cdot |\{f \in F_t : D_F > d_F\}|$

$d[k] := -(I - \frac{1}{|S_i|}\mathbf{1})\nabla L_i(w_t[k])$

if $d[k] \neq 0$ **then**

get optimal $\nu[k]$ by Alg. 2

$w[k+1] := w[k] + \nu[k] \cdot d[k]; \quad k := k+1$

else return $w[k]$

Algorithm 2: Modified Line Search (Polyline Search)

Input: $w[k], d[k], \epsilon$, n_s, ν_{maxstep}, for each task $t : \lambda_t$

$\Phi := S_i$

d_Φ : vector of coordinates of $d[k]$ corresponding to free (unblocked) tasks

$\delta := 0$; $w := w[k]$; $\mathcal{M} := \emptyset$

while $\delta < \nu_{maxstep}$ *and* $|\Phi| \geq 2$ **do**

$\nu_{\text{inc}} := \min\{{}^{1-w_t}/_{d_t} | d_t > 0, \ t \in \Phi\}$

$\nu_{\text{wless}} := \min\{{}^{w_t}/_{-d_t} | d_t < 2\epsilon, \ t \in \Phi\}$

$\nu_{\text{pdec}} := \min\{{}^{w_t}/_{-2d_t} | d_t < 0, \ t \in \Phi\}$

$\nu_{\text{max}} := \min\{\nu_{\text{inc}}, \nu_{\text{pdec}}, \nu_{\text{wless}}, \nu_{\text{maxstep}} - \delta\}$

add to $\mathcal{M}$ this: $\arg\min\{L_i(w_k + \nu \cdot d_k) | \nu \in [0, \nu_{\text{max}}]\}$

remove blocked tasks from Φ

recalculate d_Φ, $d_\Phi := (I_{|\Phi|} - \frac{1}{|\Phi|}1_\Phi)d_\Phi$

$\delta := \delta + \nu_{\text{max}}$; $w := w + \nu_{\text{max}}$

return $\arg\min \mathcal{M}$

more we optimize for the delay. There is no penalty when the schedule complies with the delay-SLO and the penalty rapidly increases with infeasibility. Let the new objective function $L(w)$ be the sum of the latest objective (10) and, for all flows f, the newly introduced penalty $P(f)$. Finally, based on the following Claim 1, we can and will rewrite (12) into equality form.

Claim 1. *We can rewrite* (12) *with equality without modifying the optimum of* $L(w)$.

The proof of Claim 1 is relegated to Appendix B. The final simplified model for answering Problem 1 is as follows, with the objective function denoted by $L(w)$:

$$\min \left(\alpha \sum_{f \in F} \max \left[0, \sum_{t: f \in F_t} \frac{\theta_t Q}{T_{i_t} w_t} - D_f \right] - \sum_{i \in [1,S]} \sum_{t \in S_i} \lambda_t w_t \right) \tag{14}$$

$$\sum\nolimits_{t \in S_i} w_t = 1, \ \forall i \in [1, S]; \qquad w_t \geq 0, \ \forall t \in S_i \tag{15}$$

B. A Model-predictive Scheduler Controller

In this section, we discuss an optimization algorithm to solve the simplified system model (14)–(15). Our optimization algorithm will apply the projected gradient method, using the (negative) gradient of the objective:

$$(\Delta w)_t = -\frac{\partial L(w)}{\partial w_t} = \alpha \sum\nolimits_{f \in F_t : d_f > D_f} \frac{\theta_t Q}{T_{i_t}} \frac{1}{w_t^2} + \lambda_t.$$

Here, θ_t can be measured from the data plane as the total CPU consumption of task t divided by the total input packet rate r_t (thus, we do not need flow delay d_f and flow-rate r_f). In addition, let $\lambda_t \in \{0, 1\}$ be a binary parameter accounting for the queue size at t. We set λ_t as follows: if there exists $f \in F_t : r_{t,f} \leq (1-\delta)R_f$ (where e.g. $\delta = 0.01$ is a tolerance) and the queue size for t is above the high-watermark then we set $\lambda_t = 1$; otherwise if the queue size for t is below the low-watermark we set $\lambda_t = 0$. Observe that $(\Delta w)_t$ is non-negative for any task t. Intuitively, each task is "greedy", constantly requesting more CPU share to process more packets with lower latency. Allowing less critical tasks to give up CPU share, for each worker i, we project the gradients of tasks assigned to i into hyperplane $\sum_{t \in S_i} (\Delta w)_t = 0$ and perform a line search.

We can now utilize the convergent version of Rosen's projected gradient method [33, pp. 593-601] to solve the model. In Alg. 1, the outer cycle ensures an iterative updating of the weights w along the projected gradient. Here, for any column n-vector x, $y = Px = (\mathbf{I} - \frac{1}{n}\mathbf{1})x$ is an orthogonal projection of x to the hyperplane $1^T x = 0$, where 1^T is a row n-vector of all ones and $\mathbf{1}$ is an $n \times n$ matrix of all ones. The modified line search (*polyline search*) along the projected gradient is detailed in Alg. 2. Here, the maximum step size ν_{max} may be a static constant. Constant ν_{inc} is introduced to ensure that no weight exceeds 1, while ν_{wless} re-ensures that no weight drops below 0 to conform to (15). In fact, ν_{wless} ensures that no weight drops below ϵ (a small positive constant), which enforces the intuitive observation that letting $w_t \sim 0$ the delay of the flows traversing t would skyrocket. Value ν_{pdec} plays a similar role: to prevent overly steep dynamics, saves half of the weight of each task in each iteration. Let F_i denote the set of flows traversing worker i and let n_s denote the number of equidistant points the line search visits. Lemma 2 shows that our optimization algorithm in each iteration either claims optimality, or departs towards the optimum.

Lemma 2. *In each step, Alg. 1 either terminates at a KKT point or else it generates an improving feasible direction. The time complexity of each step is* $O\left(|S_i|^2 |F_i| \cdot n_s\right)$.

The proof of Lemma 2 is relegated to Appendix C. Note that if there are multiple workers, then the total complexity of a control loop is $O\left(n_s \cdot \sum_{i \in [1,S]} |S_i|^2 |F_i|\right)$.

C. Dynamic Resource Allocation

So far, we assumed that the allocation of tasks to CPUs (i.e., S_i) is fixed. However, a static task allocation can easily become suboptimal with a change in the input rates or SLOs. Unfortunately, as the claim below shows, the dynamic resource allocation problem posed in Problem 2 is intractable.

Claim 2. *It is NP-hard to decide whether there exists an allocation of tasks to workers that enables meeting the rate and delay SLOs.*

The proof of Claim 2 is relegated to Appendix C. Note that if there are multiple workers, then the total complexity of a control loop is $O\left(n_s \cdot \sum_{i \in [1,S]} |S_i|^2 |F_i|\right)$.

We propose a dynamic task-reallocation heuristic to tackle intractability. In each iteration, it performs the following steps to migrate a task from an overused CPU to an underutilized one: (1) wait until the system gets stabilized; (2) detect underused and overused CPUs; (3) choose a task on an overused CPU to be migrated; (4) choose an underused CPU that has enough free capacity to run the task; and finally (5) reallocate the task. Here, a CPU i is underused if, for all $t \in S_i$, $\lambda_i = 0$, otherwise it is overused. Let $C_t = \sum_{f \in F_t} r_{t,f}\tau_{t,f}$ be the CPU usage of task $t \in S_i$ [cycle/sec], and let $D_i = T_i - \sum_{t \in S_i} C_t$ be the spare capacity on an underused CPU i [cycle/sec]. Then, task t can be moved from an overused CPU j to an underused CPU i if $C_t \leq D_i$. We note that all required parameters can be easily measured from the running system. We embed the above task migration step into various heuristic strategies for choosing the task to be moved: 1) *greedy*: choose the "most expensive" task that can be moved; 2) *max flow-affinity*: try to move tasks traversed by each flow to the same CPU; and finally 3) *max SLO-compliance*: move candidate task traversed by the flows with the least strict delay-SLOs. In the evaluations (§VI), we combine these resource allocation heuristics with the projected gradient based controller for solving the full-fledged scheduling problem defined in Problem 2.

V. Simulator

We created a discrete time simulator to experiment with our controllers. Fig. 3 summarizes the workflow of the simulator. We initialize the simulator with the given system G, G_t, T_i, c_v and flows $f = (p_f, \rho_f, R_f, D_f), f \in F$, and we choose a set of initial task weights $w_t : t \in T$. Then, the following steps are repeated by the simulator in each iteration: (1) run the system model detailed in §II-B to produce the system state and then (2) run the optimizer to compute optimal w_t with respect to the system state. Given scheduling weights w_t for each task

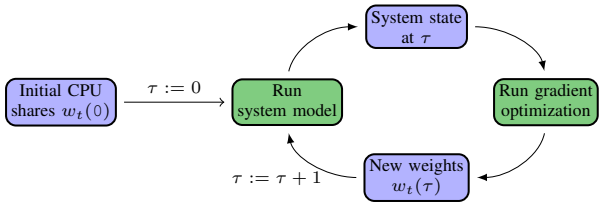


Figure 3. Simulator Workflow.

t, the state of the system can be obtained as follows. First, compute per-task packet rates $r_{t,f}$ by solving the following linear program:

$$\max \sum_{f \in F} r_f \tag{16}$$

$$\sum_{f \in F_t} r_{t,f}\tau_{t,f} \leq w_t T_{i_t} \qquad t \in S_i, i \in [1, S] \tag{17}$$

$$r_{s,f} = r_{t,f} \qquad f \in F, (s,t) \in \pi_f \tag{18}$$

$$r_f = r_{s_f,f} \qquad f \in F \tag{19}$$

$$0 \leq r_f \leq \rho_f, r_{t,f} \geq 0 \qquad t \in T, f \in F_t. \tag{20}$$

Then, determine the per-packet task delays: $\theta_t = \frac{\sum_{f \in F_t} r_{t,f}\tau_{t,f}}{\sum_{f \in F_t} r_{t,f}}$. Finally, compute the flow delays: $d_f = \sum_{t: f \in F_t} \left(\max \left\{ \max_{t' \in S_i : t' \neq t} \left(B \frac{\theta_{t'}}{T_{i_t}} \right), \frac{Q}{B} \frac{B\theta_t}{T_{i_t}} \frac{1}{w_t} \right\} + B \frac{\theta_t}{T_{i_t}} \right)$.

In an actual implementation, the "back-pressure" signal λ_t could be measured from the running system (recall, we have to set $\lambda = 1$ whenever the queue size would be above the threshold); unfortunately, in our simulator, we have to obtain this parameter from the system model. A task t is stressed if, given its CPU share w_t, it does not have enough compute resources to process all traffic it receives. Based on this, we use the following rule: after solving (16)–(20), set $\lambda_t = 1$ for $t \in T$ exactly if task t utilizes all its CPU share $w_t T_{i_t}$: $\sum_{f \in F_t} r_{t,f}\tau_{t,f} \geq (1-\delta) w_t T_{i_t}$, and there is demand for more traffic: $\min(R_f, \rho_f) \geq (1-\delta) r_f$, where δ is again a tolerance, e.g., $\delta = 0.01$.

At this point, we have all the parameters available to run the model: we execute *a single step of the projected gradient algorithm* (see Alg. 1) followed by *a single line-search* (see Alg. 2) to obtain the new weights. The simulator then goes back to obtaining the system state with respect to the new scheduling weights, and this loop is repeated until the total system time exceeds a given limit. We implemented the simulator in Python; the code is available for download at [19].

VI. Evaluation

We evaluated our real-time scheduler controller logic in extensive simulation studies. Since the model critically depends on the delay estimate (§III-B), first we confirm this estimate on a real software switch. Then, we present a detailed numerical evaluation of our controller logic (§IV-B) with the simulator and finally we benchmark our dynamic resource allocation scheme.

To the best of our knowledge, currently, there is no official benchmark suite available to test our control loops. Therefore, we will use synthetic and real-life sample pipelines from [30], [32] taken from a 5G benchmark suite [4] for the evaluations. In particular, for the synthetic tests, we chose the *fork* example pipeline of Fig. 2 and a *taildrop* pipeline consisting of 3 tasks, from which the last one is heavyweight [30]. As a realistic pipeline, we chose the 5G Mobile Gateway (*MGW*) from [4]. The pipelines are originally implemented in BESS, then converted to our simulator. We assume workers have unit speed, we set $B = Q = 1$, and we let the line search to make $n_s = 5$ tries at each line segment with a maximum step size ν_{maxstep} chosen as 0.01 or 0.025.

A. Validating the Delay Estimate

A dependable delay estimation is crucial for scheduling latency-sensitive pipelines. Hence, our evaluation starts with the validation of the task delay estimate of §III-B. For this purpose, we implement the fork example pipeline of Fig. 2 in a widely-deployed software switch: BESS [18], [34]. We adjust the following parameters: *i)* the fan-out of Task0 (the number of tasks connected to the egress module of Task0); *ii)* the execution time of Task1; and *iii)* the ratio of weights between Task1 and other egress tasks (e.g., Task2).

Fig. 4 shows the condensed results (delay estimate and measured delay percentiles) with two egress tasks (Task1 and Task2) and task execution times varying in 100, 10k, and 10M CPU cycles as Task1/Task2 scheduling weight ratio is set

between $(0, 1]$. We find that, except for extreme Task1/Task2 weight ratios, the delay estimate coincides with the 99-th percentile measured delay, with a slight tendency for the model to overshoot the delay. This confirms that *our delay estimate is a good fit to drive the real-time scheduling control loop*.

B. Control Algorithm

Our first round of evaluations focuses on establishing the viability of the control algorithm and understand the effect of choosing the optimization parameter α, which, recall, decides whether the scheduler will favor satisfying the delay-SLOs at the cost of potentially violating the rate-SLOs (α large) or the *vice versa* (α small). Then, we will run the model on more complex pipelines to understand the control dynamics.

First, we tested our controller on the *fork* pipeline of Fig. 2. We deliberately set the SLOs so that there is no way for the controller to satisfy all: this stresses the controller to the extreme and allows us to observe the effect of choosing α. Our results are summarized in Fig. 5. First, we observe that *the controller chooses the task scheduling weights so that in each step the system is driven closer to the SLOs*. This justifies the basic viability of the model. Second, as it was expected *the lower the value of α the more the controller favors fulfilling the rate-SLOs at the cost of violating the delay requirements*: for $\alpha = 0.05$ the rate-SLO of both flows and the delay-SLO of the first flow are all satisfied but the delay-SLO of the other flow is violated, while for $\alpha = 1$ the delay-SLOs all hold (with a small error for the first flow) but one of the rate-SLOs is violated. In the context of O-RAN delay-SLOs are more important; correspondingly in the below we will use the setting $\alpha = 1$ (i.e., favor delay at the cost of rate). Note that we found an SLO violation in all examples; this is because, recall, we deliberately set non-fulfillable SLOs. Re-running the evaluations with looser SLOs *we found that our control algorithm can always drive the system to an SLO-compliant state* in just a couple of iterations (results not shown here). Interestingly, we find the same "sawtooth pattern" in the control action that is well-known in typical online controllers [35].

We repeated the benchmark on the *taildrop* pipeline with a single flow. Recall, this pipeline comprises a chain of 3 tasks, the last being the most expensive. Consequently, an equal weight setting will be suboptimal: the last heavyweight task will not get enough CPU share to run the costly processing on all packets fed to it by the preceding lightweight tasks, causing a so called *taildrop* phenomenon where we spend significant resources processing traffic just to drop it at a later stage in the pipeline [30]. Clearly, in order to remedy this the scheduling weight of the last task needs to be increased. Fig. 6a shows that this is exactly what our controller does: starting from approximately identical initial task weights it rapidly scales up the scheduling weight of the heavy task (Task 2) and decreases the weight of the other tasks. Eventually, at the 14-th iteration all SLOs are met and internal packet drop disappears (this can be tracked from observing the queue size signal: when $\lambda = 1$ there is a task input queue that is filled to capacity and drops packets).

The last evaluation was performed on a mobile gateway packet-processing (*MGW*) pipeline (Fig. 7) taken from the official 5G NFV benchmark suite [4]. A 5G mobile gateway connects a set of mobile user equipments to the public Internet. This requires a complex pipeline with differentiated traffic classes (called "bearers" here for simplicity). Traffic flows either in uplink or downlink direction, and is further classified among bearers. Users may have connections on multiple bearers both in the uplink and downlink direction, and each user's connection is considered a separate flow. In our evaluations, bearer0 (both uplink and downlink) represents mobile voice and multimedia traffic with firm performance requirements, while the rest of the bearers represent bulk traffic. The number of concurrent flows is $2\times$ the number of users on the first bearer (bearer0) due to separate uplink and downlink connections. The number of bearers, users, and users of the voice/multimedia bearers (bearer0 users) as well as the capacity and the number of CPUs are parameters.

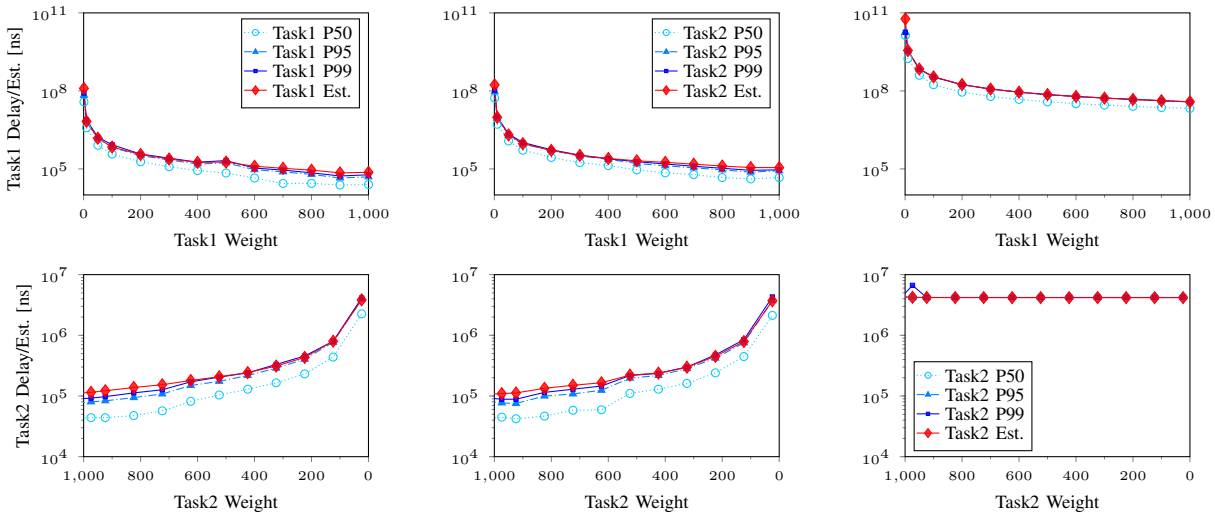
Fig. 6b shows results on an MGW pipeline with 37 modules organized into 5 tasks and 3 workers: the `ingress` and `egress` tasks both have a separate worker, while the `bearer dl 1`, `bearer ul 1` and `bearer 0 ul user 0` tasks share a common worker (see the full description in our GitHub repository [19]). Our findings for this complex benchmark are similar as before: in just about a dozen iterations *the controller settles the system in a fully SLO-compliant state and completely removes internal packet drop*.

C. Dynamic Resource Allocation

In the context of this work, resource allocation means decomposing the dataflow graph that represents the processing pipeline into separate subgraphs, each representing a basic scheduling unit (i.e., task). So far we have experimented with static resource allocations; in this evaluation round we benchmark our dynamic resource allocations schemes.

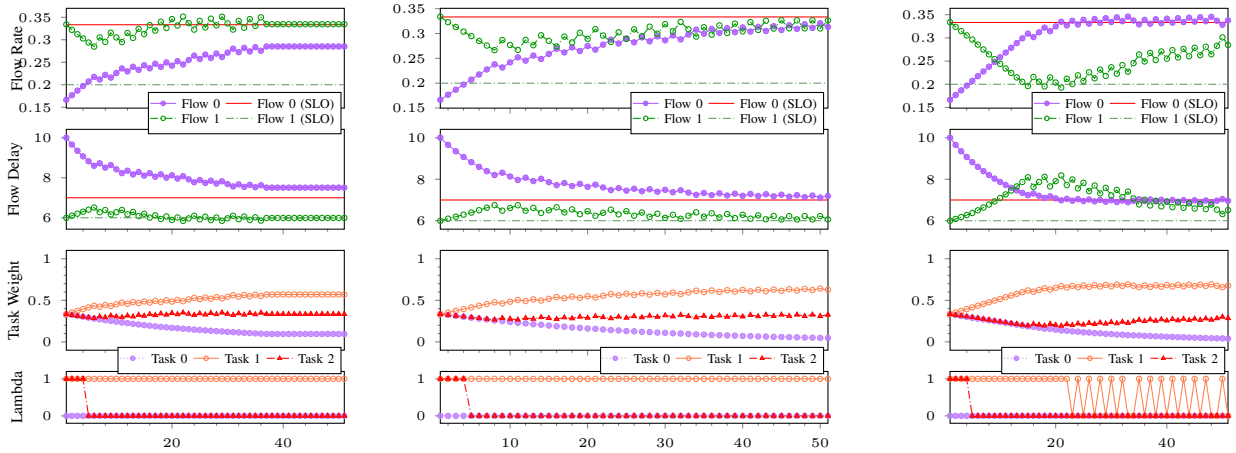
First we experiment with the *taildrop* pipeline, but this time setting tight SLOs so that the system does not have enough resources to fulfill the SLOs. Fig. 8a shows what happens after we open a new worker at iteration 10 and enable the greedy task reallocation scheme. Not surprisingly, the greedy resource re-allocation scheme quickly moves the heavy task from the over-utilized worker to the new (underutilized) CPU, driving the system to an SLO-compliant state in a single step. Fig. 8b shows a benchmark where the module costs and SLOs are set so that further optimization is needed to achieve SLO compliance after the task reallocation, and Fig. 8c shows the same test on the *fork* pipeline (again enabling a new worker at step 10). Our observations are similar as before: our scheduler control loops quickly drive the system to an optimal state.

Finally, we evaluated the dynamic resource allocation scheme on more complex examples. Fig. 8d shows the results on the *taildrop* pipeline, but this time with 7 processing-heavy tasks that together greatly overload the (single) initial worker. To remedy this, in the 10-th step, we add a new worker to the system. We observe that the greedy task migration strategy is effective in this case too: by moving tasks across workers it quickly removes SLO-violation. Eventually, in step 72 all SLOs are satisfied and internal packet drops disappear, *confirming that our combined resource allocation and scheduling strategy*



(a) Fast Task 1 (b) Equal Execution Times (c) Slow Task 1

Figure 4. Validating Delay Estimate: Example Pipeline (Fig. 2) implemented in BESS with Task 2 execution taking 10k CPU cycles while Task 1 execution time varies: (a) 100, (b) 10k, and (c) 10M CPU cycles. Note: task weights on x-axes follow the BESS weights notation where $w_t = 1$ requires 1000 units.



(a) Large ($\alpha = 1.0$) (b) Medium ($\alpha = 0.2$) (c) Small ($\alpha = 0.05$)

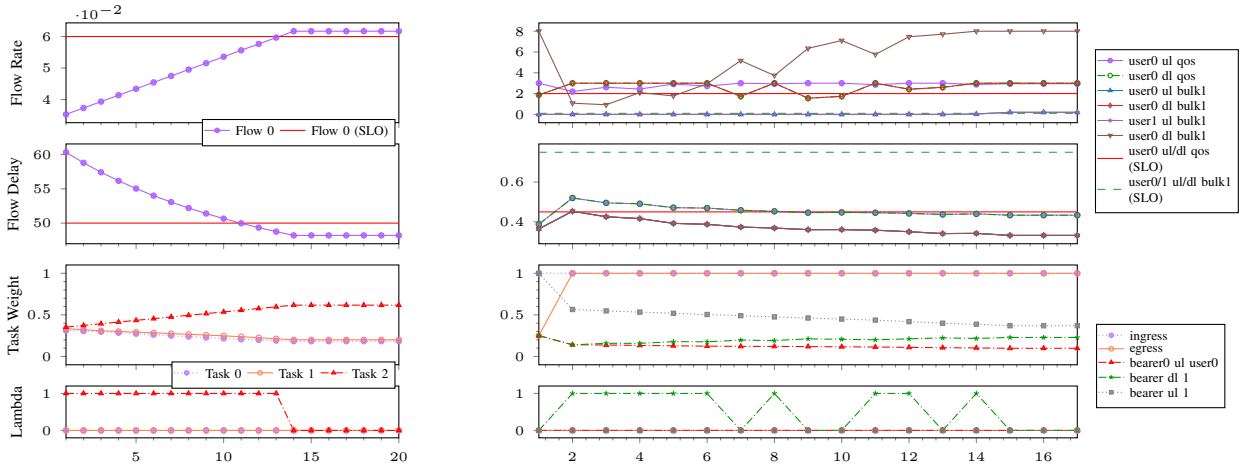
Figure 5. Effect of parameter α responsible for weighing in the possible delay SLO violations, measured on the *fork* pipeline.

can rapidly find the system optimum. We also repeated the benchmark in the *MGW* pipeline, see Fig. 8e. This time we did not add a new CPU, we just enabled task migration at step 10. At this point the controller reallocated the `bearerul1` task to the worker of the `ingress` task, decreasing the number of control periods needed to achieve SLO compliance from 15 to 12 steps compared to the fixed resource allocation setting.

VII. Related Work

VNF Performance Prediction: Running multiple NFs on a single host leads to performance degradation due to contention in shared hardware resources such as last-level CPU cache [36], [37] or packet I/O [6]. Performance prediction of VNFs is therefore crucial for guaranteeing SLOs. SLOMO [6] predicts collocated VNF performance using machine learning. Bolt [5] leverages symbolic execution to estimate the processing cost of different traffic classes processed by a single VNF; also generalized to NF chains too. In addition, [38] presents a discrete Markovian queuing model and [39] presents a discrete-time model for a single-queue single-server system with known service-time distribution. As opposed to our work, most of these methods require extensive profiling of NFs. Recall, we relax the requirement of known module processing costs in §IV.

Meeting SLOs in NFV Platforms: Besides high performance, meeting SLOs is a highly-desired behavior of NFV platforms. Grus [9], an NFV framework with GPU offload, introduces a multi-layer system with admission control and delay prediction model to guarantee delay-SLOs. As opposed to our work, Grus guarantees delay-SLO only for single VNF deployments, and the model is tailored for the GPU offloading scenario. In contrast to Grus, ResQ [37] provides performance isolation at CPU last-level cache solving the noisy neighbor problem of VNFs, and enables enforcing SLOs. NFV-RT [40] provides



(a) Taildrop Pipeline on 3 Nodes
Task costs: Task0 & Task1: 1,Task2: 10.

(b) MGW Pipeline: single user on QoS and Bulk bearers (uplink/downlink).

Figure 6. Effect of the controller on pipelines *taildrop* and *MGW* accompanied with simultaneously satisfiable delay and rate SLOs, respectively. With $\alpha = 1$, the controller found a feasible solution in 15 control periods in both cases.

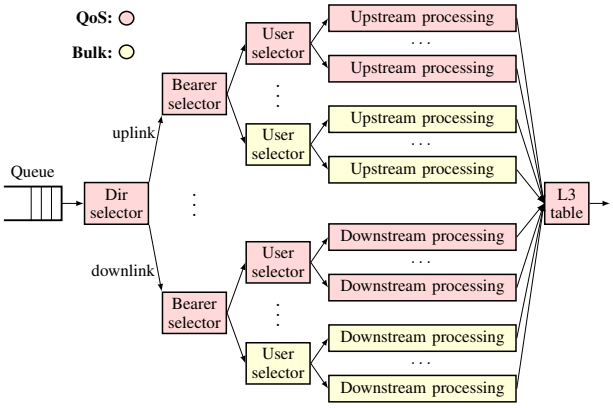


Figure 7. Mobile Gateway (MGW) Pipeline.

soft real-time guarantees for NF service chains deployed in data center environment using a fat-tree topology. Batchy [32] is a general-purpose dataflow graph scheduler framework, which enables enforcing delay-SLOs at the millisecond time scale even at multiple millions of packets per second of throughput. To achieve this, Batchy uses controlled queuing to efficiently reconstruct fragmented batches in accordance with strict SLOs. However, Batchy considers only delay-SLOs and it is restricted to a single task on a single worker (but see also [31]). Quadrant [29] enables FaaS abstractions in NFV workloads, and provides a performance-aware scheduler. Quadrant adaptively controls number of batches to process at a single poll to satisfy SLOs while minimizing context switch overhead coming from the containerized environment of the NF runtime, and supports delay-SLO-based scaling of NF-chains. As oppose to our work, Quadrant is a complete NFV platform. As opposed to these works, our closed loop scheduler is built on an *analytical system model*, without relying on static performance benchmarks or costly prior AI/ML training. In addition, our controllers can handle both rate- and delay-type SLOs.

VIII. Conclusions

In this paper, we present a controller framework for real-time execution of disaggregated services on commercial off-the-shelf hardware. The controller relies on a comprehensive analytical model, combining the formal model with monitoring data to find an optimized schedule rapidly. We present a model-based gradient-optimization control algorithm to provide optimal scheduling. We evaluate our model on a set of synthetic and realistic pipelines. Our results show that the model predicts the delays reliably, and the control algorithm can converge the system to an SLO-compliant state in just a few iterations, potentially performing dynamic resource allocation when SLOs can not be met otherwise. Future work involves dynamically adjusting parameters such as controller aggressiveness, and extending the model to multiple switches.

References

[1] M. Polese, L. Bonati, S. D'Oro, S. Basagni, T. Melodia, Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges, Commun. Surveys Tuts. 25 (2) (2023) 1376–1411. doi:10.1109/COMST.2023.3239220.
URL https://doi.org/10.1109/COMST.2023.3239220

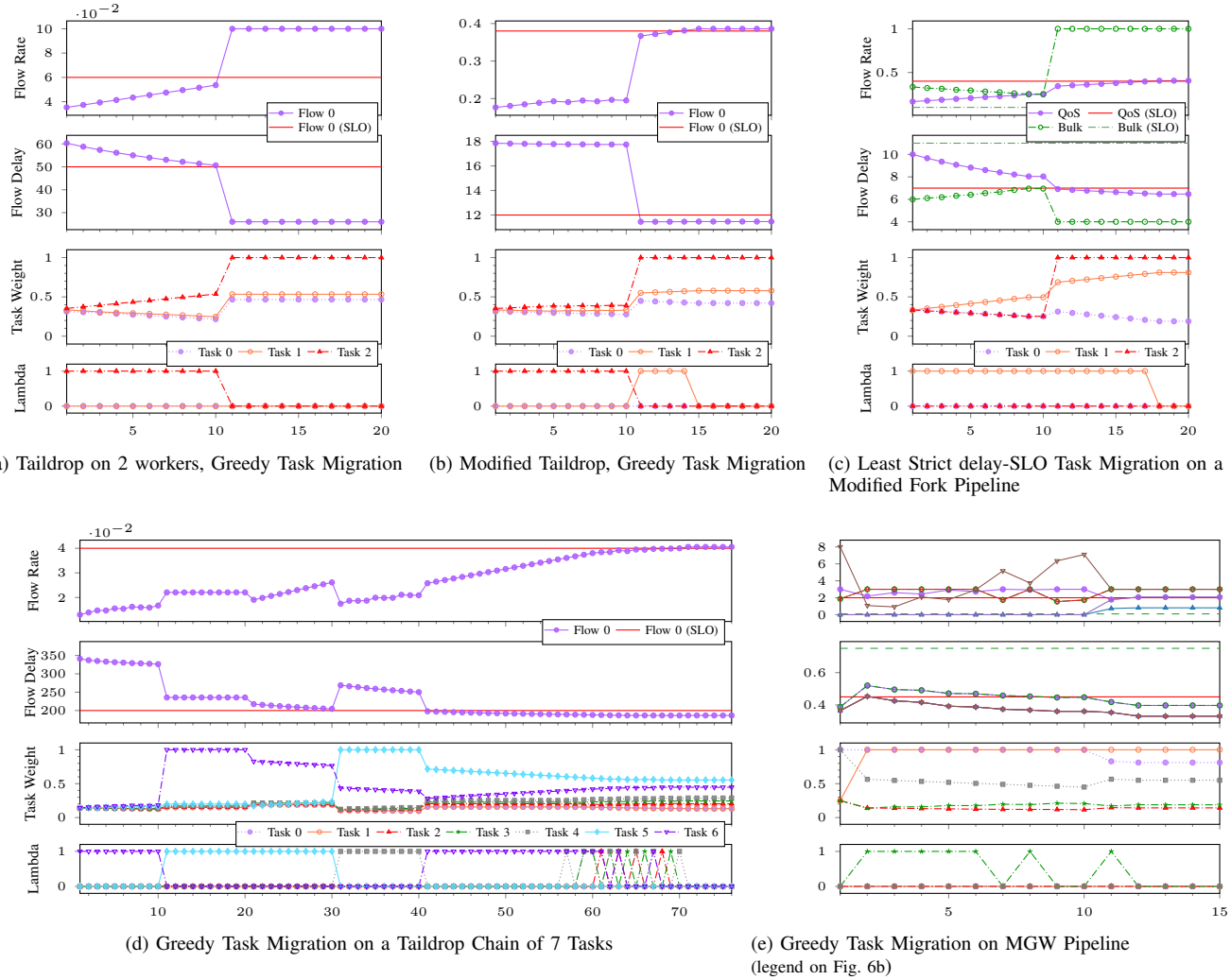
[2] F. Voigtländer, A. Ramadan, J. Eichinger, C. Lenz, D. Pensky, A. Knoll, 5G for robotics: Ultra-low latency control of distributed robotic systems, in: 2017 International Symposium on Computer Science and Intelligent Controls (ISCSIC), IEEE, 2017, pp. 69–72.

[3] NGMN Alliance, 5G white paper, Next generation mobile networks, white paper (2015) 1–125.

[4] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, G. Rétvári, The Price for Programmability in the Software Data Plane: The Vendor Perspective, IEEE Journal on Selected Areas in Communications 36 (12) (2018) 2621–2630.

[5] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, G. Candea, Performance Contracts for Software Network Functions, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 517–530.
URL https://www.usenix.org/conference/nsdi19/presentation/iyer

[6] A. Manousis, R. A. Sharma, V. Sekar, J. Sherry, Contention-Aware Performance Prediction For Virtualized Network Functions, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 270–282.



(a) Taildrop on 2 workers, Greedy Task Migration

(b) Modified Taildrop, Greedy Task Migration

(c) Least Strict delay-SLO Task Migration on a Modified Fork Pipeline

(d) Greedy Task Migration on a Taildrop Chain of 7 Tasks

(e) Greedy Task Migration on MGW Pipeline (legend on Fig. 6b)

Figure 8. Task migration results. The reallocation controller runs in every 10-th control period.

doi:10.1145/3387514.3405868.
URL https://doi.org/10.1145/3387514.3405868

[7] P. Martin, Scaling an Application, Apress, Berkeley, CA, 2021, Ch. Scaling an Application, pp. 73–78. doi:10.1007/978-1-4842-6494-2_7.
URL https://doi.org/10.1007/978-1-4842-6494-2_7

[8] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, J. Sherry, Achieving 100Gbps Intrusion Prevention on a Single Server, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 1083–1100.

[9] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, J. Wu, Grus: Enabling Latency SLOs for GPU-Accelerated NFV Systems, in: IEEE ICNP, 2018, pp. 154–164.

[10] H. N. Schuh, W. Liang, M. Liu, J. Nelson, A. Krishnamurthy, Xenic: SmartNIC-Accelerated Distributed Transactions, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 740–755. doi:10.1145/3477132.3483555.
URL https://doi.org/10.1145/3477132.3483555

[11] M. Polese, L. Bonati, S. D'Oro, S. Basagni, T. Melodia, ColO-RAN: Developing Machine Learning-based xApps for Open RAN Closed-loop Control on Programmable Experimental Platforms, IEEE Transactions on Mobile Computing (2022) 1–14doi:10.1109/TMC.2022.3188013.

[12] X. Wang, J. D. Thomas, R. J. Piechocki, S. Kapoor, R. Santos-Rodríguez, A. Parekh, Self-play learning strategies for resource assignment in Open-RAN networks, Computer Networks 206 (2022) 108682. doi:https://doi.org/10.1016/j.comnet.2021.108682.
URL https://www.sciencedirect.com/science/article/pii/S138912862100551X

[13] D. Bankov, E. Khorov, A. Lyakhov, M. Sandal, Enabling real-time applications in wi-fi networks, International Journal of Distributed Sensor Networks 15 (5) (2019) 1550147719845312. arXiv:https://doi.org/10.1177/1550147719845312, doi:10.1177/1550147719845312.
URL https://doi.org/10.1177/1550147719845312

[14] N. Nikaein, Processing radio access network functions in the cloud: Critical issues and modeling, in: Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services, MCS '15, 2015, p. 36–43.

[15] N. Nikaein, E. Schiller, R. Favraud, R. Knopp, I. Alyafawi, T. Braun, Towards a Cloud-Native Radio Access Network, Springer International Publishing, 2017, Ch. Towards a Cloud-Native Radio Access Network, pp. 171–202.

[16] W. Azariah, F. A. Bimo, C.-W. Lin, R.-G. Cheng, R. Jana, N. Nikaein, A survey on Open Radio Access Networks: Challenges, research directions, and open source approaches (2022).
URL https://arxiv.org/abs/2208.09125

[17] Intel, Data Plane Development Kit, http://dpdk.org (2024).

[18] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, S. Ratnasamy, SoftNIC: A Software NIC to Augment Hardware, Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley (May 2015).
URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[19] Source Code and Artifacts on Github, https://github.com/hsnlab/realtime-nf-scheduling-in-oran (2024).

[20] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, G. Bianchi, Survey of performance acceleration techniques for Network Function Virtualization, Proceedings of the IEEE 107 (4) (2019) 746–764. doi:10.1109/JPROC.2019.2896848.

[21] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi,

High-Speed Software Data Plane via Vectorized Packet Processing, IEEE Communications Magazine 56 (12) (2018) 97–103.
[22] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek, The Click Modular Router, in: ACM SOSP, 1999, pp. 217–231.
[23] T. Barbette, C. Soldani, L. Mathy, Fast Userspace Packet Processing, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2015, pp. 5–16.
[24] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, S. Shenker, NetBricks: Taking the V out of NFV, in: USENIX OSDI, 2016, pp. 203–216.
[25] H.-Y. Chen, et al., TensorFlow: A system for large-scale machine learning, in: USENIX OSDI, Vol. 16, 2016, pp. 265–283.
[26] W. Taymans, S. Baker, A. Wingo, R. S. Bultje, S. Kost, GStreamer Application Development Manual, Samurai Media Limited, United Kingdom, 2016.
[27] C. A. Waldspurger, W. E. Weihl, Stride scheduling: Deterministic proportional share resource management, Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.
URL http://www.waldspurger.org/carl/papers/stride-mit-tm528.pdf
[28] C. Lan, An Architecture for Network Function Virtualization, Ph.D. thesis, UC Berkeley (2018).
[29] J. Wang, T. Lévai, Z. Li, M. A. M. Vieira, R. Govindan, B. Raghavan, Quadrant: A Cloud-Deployable NF Virtualization Platform, in: Proceedings of the 13th Symposium on Cloud Computing, SoCC '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 493–509. doi:10.1145/3542929.3563471.
URL https://doi.org/10.1145/3542929.3563471
[30] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, X. Fu, NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains, in: ACM SIGCOMM, 2017, pp. 71–84.
[31] T. Lévai, G. Rétvári, Batch-scheduling Data Flow Graphs with Service-level Objectives on Multicore Systems, INFOCOMMUNICATIONS JOURNAL 14 (2022) 43–50. doi:10.36244/ICJ.2022.1.6.
[32] T. Lévai, F. Németh, B. Raghavan, G. Rétvári, Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives, in: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), USENIX Association, Santa Clara, CA, 2020, pp. 633–649.
URL https://www.usenix.org/conference/nsdi20/presentation/levai
[33] M. S. Bazaraa, H. D. Sherali, C. M. Shetty, Nonlinear programming: theory and algorithms, John Wiley & Sons, 2013.
[34] O. N. Foundation, Open Mobile Evolved Core, https://opennetworking.org/omec/ (2024).
[35] D. Bansal, H. Balakrishnan, S. Floyd, S. Shenker, Dynamic behavior of slowly-responsive congestion control algorithms, ACM SIGCOMM Computer Communication Review 31 (4) (2001) 263–274.
[36] M. Dobrescu, K. Argyraki, S. Ratnasamy, Toward Predictable Performance in Software Packet-Processing Platforms, in: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), USENIX Association, San Jose, CA, 2012, pp. 141–154.
URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu
[37] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, S. Shenker, ResQ: Enabling SLOs in Network Function Virtualization, in: USENIX NSDI, 2018, pp. 283–297.
[38] Z. Su, T. Begin, B. Baynat, Towards including batch services in models for DPDK-based virtual switches, in: GIIS, 2017, pp. 37–44.
[39] S. Lange, L. Linguaglossa, S. Geissler, D. Rossi, T. Zinner, Discrete-Time Modeling of NFV Accelerators that Exploit Batched Processing, in: IEEE INFOCOM, 2019, pp. 64–72.
[40] Y. Li, L. T. Xuan Phan, B. T. Loo, Network functions virtualization with soft real-time guarantees, in: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, 2016, pp. 1–9.
[41] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., USA, 1979.

Tamás Lévai received the MSc in Computer Engineering at the Budapest University of Technology and Economics (BME) in 2016. He received his PhD degree in Informatics in 2022 at BME. Currently, he is an Assistant Lecturer at BME, and a Research Fellow at HUN-REN TKI. His research interest focuses on computer networks and distributed computing, mainly software-defined networking, cloud native computing and real-time communications.

Balázs Vass received his MSc degree in applied mathematics at ELTE, Budapest in 2016. He finished his PhD in informatics in 2022 on the Budapest University of Technology and Economics (BME). His research interests include networking, survivability, combinatorial optimization, and graph theory. He was an invited speaker of COST RECODIS Training School on Design of Disaster-resilient Communication Networks '19. He is a TPC member of IEEE INFOCOM '23, '24, and '25.

Gábor Rétvári received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology and Economics in 1999 and 2007, where he is now a Senior Research Fellow. His research interests include all aspects of network routing and switching, the programmable data plane, and the networking aspects of cloud native computing. He is a co-founder and CTO of L7mp Technologies, a company specializing in running large-scale WebRTC services over Kubernetes.

APPENDIX

A. Proof of Lemma 1

Proof: We start with the notations of the stride scheduling paper [27], with time quantum and stride scaled to: $\texttt{quanta} = 1$, $\texttt{stride}_1 = 1$. We will turn to our notations meanwhile, with: $\texttt{tickets}(t) = w_t$, $\texttt{quanta}(t) = \theta_t$. By definition, for any task $t \in S_i$, $\texttt{Pass}(t) = \texttt{\#scheduled}(t) \cdot \texttt{stride}(t) \cdot \frac{\texttt{quanta}(t)}{\texttt{quanta}} = \texttt{\#scheduled}(t) \cdot \frac{\texttt{stride}_1}{\texttt{tickets}(t)} \cdot \frac{\texttt{quanta}(t)}{\texttt{quanta}} = \texttt{\#scheduled}(t) \cdot \frac{\theta_t}{w_t}$ (Eq. Pass). Also, after a long time $\frac{\texttt{Pass}(t)}{\texttt{Pass}(t')} \to 1$, for any $t' \in S_i$ (Eq. infty). Furthermore, the timeshare a task $t \in S_i$ gets (on server i) is:

$$\frac{\texttt{\#scheduled}(t) \cdot \frac{\texttt{quanta}(t)}{\texttt{quanta}}}{\sum_{t' \in S_i} \texttt{\#scheduled}(t') \cdot \frac{\texttt{quanta}(t')}{\texttt{quanta}}} =$$

$$= \frac{\texttt{\#scheduled}(t) \cdot \theta_t}{\sum_{t' \in S_i} \texttt{\#scheduled}(t') \cdot \theta_{t'}} \overset{\text{Eq. Pass}}{=}$$

$$= \frac{\texttt{Pass}(t) \cdot w_t}{\sum_{t' \in S_i} \texttt{Pass}(t') \cdot w_{t'}} \overset{\text{Eq. infty}}{\to} \frac{w_t}{\sum_{t'} w_{t'}} = \frac{w_t}{1} = w_t.$$

Based on this, the time needed to accumulate the amount of time for task t to run once is $\frac{\theta_t}{w_t}$. ∎

B. Proof of Claim 1

Proof: Suppose indirectly that there is no optimal solution where the task weights on each worker add up to 1. Take an optimal solution w of (14). For each worker i, assign new weights for the tasks of i: $w'_t := {}^{w_t}/_{\sum_{t \in S_i} w_t}$, for all $t \in S_i$. Observe that the new task weights add up to 1 on each worker. Also, weights w'_t are strictly greater than the old weights w_t, thus the objective function value either decreases or stays the same, since: $(\Delta w)_t = -\frac{\partial L(w)}{\partial w_t} = \alpha \sum_{f \in F_t : d_f > D_f} \frac{\theta_t Q}{T_{i_t}} \frac{1}{w_t^2} + \lambda_t \geq 0$. The former two observations yield a contradiction. ∎

C. Proof of Lemma 2

To prove this Lemma, we need the following Claim:

Claim 3. *Given a worker, suppose every task of the worker has a strictly positive cost. Then, given any $w_t \geq 0$ s.t. $\sum_t w_t = 1$, the new weight vector w'_t yielded by Rosen's method (like Alg. 1) is strictly positive (i.e., $w'_t > 0$ s.t. $\sum_t w'_t = 1$).*

Algorithm 3: Rosen's Projected Gradient Method

Input: A, b, Q, q, f, and x_1 s.t. $Ax_1 \leq b, Qx_1 = q$
Find: $\arg\min\{f(x)|Ax \leq b, Qx = q\}$
1 $k := 1$
2 Divide A and b into A_1, A_2 and b_1, b_2, resp.; s.t.: $A_1 x_k = b_1$, $A_2 x_k < b_2$
3 // 1 -main loop
4 $M := \begin{bmatrix} A_1 \\ Q \end{bmatrix}$
5 **if** M *is vacuous* **then**
  if $\nabla f(x_k) = 0$ **then**
    return x_k
  else
    $d_k := -\nabla f(x_k)$
    GOTO 2
else
  $P := I - M^t(MM^t)^{-1}M$
6 $d_k := -P\nabla f(x_k)$
  if $d_k \neq 0$ **then**
    GOTO 2
  else
7 $z := \begin{bmatrix} u \\ v \end{bmatrix} := -(MM^t)^{-1}M \cdot \nabla f(x_k)$
8 **if** $u \geq 0$ **then**
      return x_k (KKT point) and z (associated Lagrange multipliers)
    else
      let j be an index s.t.: $u_j < 0$; delete j^{th} row of A_1
9 GOTO 1
// 2 -line search
if $d_k \leq 0$ **then**
  $\lambda_{\max} := \infty$
else
  $\hat{b} := b_2 - A_2 x_k$; $\hat{d} := A_2 d_k$
10 $\lambda_{\max} := \min\{\hat{b}_i/\hat{d}_i : \hat{d}_i > 0\}$
$\lambda_k := \arg\min\{f(x_k + \lambda \cdot d_k)|\lambda \in [0, \lambda_{\max}]\}$
11 $x_{k+1} := x_k + \lambda_k d_k$
12 Divide A and b into A_1, A_2 and b_1, b_2, resp., s.t.: $A_1 x_k = b_1$, $A_2 x_k < b_2$
13 $k := k + 1$; GOTO 1

Proof: The penalty function has to be defined to be $+\infty$, if a task with positive cost has zero weight (the delays of the flows through it are infinite, thus, they would be present in the new penalty function too). Suppose for a task t that $w_t \searrow 0$ (meaning the delays of the flows going through t are exceeding their delay SLO a lot). Then, $1/w_t \to +\infty$, thus (14) tends to infinity. ∎

Proof of Lemma 2: The convergent version of Rosen's conjugate gradient method is summarized in Alg. 3. First, we note that by [33, Thm. 10.5.7.], in each iteration, Alg. 3 indeed generates an improving direction, or terminates in a KKT point. Next, we prove that Alg. 1, being a slight variant of Alg. 3, also generates an improving direction, or terminates in a KKT point. In our case, we have the following. $A = -I, b = 0, Q = \underline{1}, q = 1$, original x_1 equaling $w[1] = [1/|S_i|, \ldots, 1/|S_i|]$, $f(x) = L(x)$. Q not being vacuous means that we never enter the 'then' branch of the if statement of Alg. 3 line 5, simplifying our algorithm. We can see that by Claim 3, A_1 is vacuous in our case. Thus, $M = Q = \underline{1}$, yielding $P = (I - \frac{1}{|S_i|}\mathbf{1})$. Suppose now that $d[k] = 0$. In line 7, vector u has dimension 0, because it is associated with matrix A_1 that is vacuous in our case. Thus, our algorithm never enters the else branch of the if statement in Alg. 3 line 8. We also do not need vector z (the Lagrange multipliers) calculated in line 7. Turning now to the second part of Alg. 3, one can check that the initializations and the first run of the while cycle of Alg. 2 indeed correspond to the line search of 3, with a slight modification of preventing task weights getting mistakenly too close to 0, that may happen because we do not recalculate the λ_t and θ_t values while performing the line search. Subsequent runs of the while cycle ensure that our control algorithm does not get stuck with minor steps by enabling to search for better solutions along the re-projected gradient, where blocked tasks were excluded. We note that because of locally simplifying the objective function (via not recalculating the λ_t and θ_t) during the search, the resulting solution $w[k+1]$ after an iteration may end up with a higher objective function value than $w[k]$.

Lastly, we turn to the complexity of an iteration of Alg 1. In Line 3, determining the gradient takes $O(|S_i| \cdot |F_i|)$ time. Determining $d[k]$ in Line 4 takes $O(|S_i|)$ since the specific structure of the matrix. Aside from the steps of Alg. 2, the remaining operations fit in this complexity. Turning to Alg. 2, we can see that the complexity of lines 1-3 is $O(s_i)$. Further, the steps in the while loop are repeated at most $|S_i|$ times, since the only way a new iteration of the loop is started is when at least one task gets blocked - and there are $|S_i|$ tasks. Line 8 takes $O(n_s \cdot |S_i| \cdot |F_i|)$, all the other tasks within an iteration of the cycle take $O(S_i)$. We can conclude that an iteration of Alg. 1 takes $O\left(|S_i|^2|F_i| \cdot n_s\right)$ time. ∎

D. Proof of Claim 2

Proof: We will show that the NP-hard PARTITION problem [41, SP12] is a special case of our problem. The PARTITION problem goes as follows. Given a set A of positive integers, can A be partitioned in sets A_1 and A_2 such that the elements of the partitions are equal?

We take an instance of the PARTITION problem with $A = \{a_1, a_2, \ldots, a_{|A|}\}$. Let $\mathcal{A} := \sum_{a_i \in A} a_i$. We take a number of $|A|$ modules $v_1, \ldots, v_{|A|}$, needing $a_1, \ldots, a_{|A|}$ (times constant) CPU cycles to process a single packet. We have a set of tasks $t_1, \ldots, t_{|A|}$. The i^{th} module is contained by task t_i, and each task contains exactly one module. The dataflow graph is a simple path $v_1, v_2, \ldots, v_{|A|}$, and there is a single flow f traversing it. The flow has infinite delay SLO, while its rate SLO is $2/\mathcal{A}$. There are two workers, each with unit time budget. Task t_i has an associated weight w_{t_i}. We note that, in this simple setting, for each task, $\theta_{t_i} = \tau_{t_i,f} = a_i$.

Without going into details, we can see, that, according to the system model (16)–(20) (and also intuitively), the rate of f will be $\min_{i \in \{1,\ldots,|A|\}} \frac{w_{t_i}}{a_i}$. This means, that for all task t_i, $w_{t_i} \geq \frac{2a_i}{\mathcal{A}}$ is needed to reach the rate SLO. This also ímeans $w_{t_i} = \frac{2a_i}{\mathcal{A}}$ for all task t_i, since if indirectly, there was a task with strictly larger weight than the right hand side, the weights would add up to more than 2, that is a contradiction.

In addition, since the task weights on each worker may add up to ≤ 1, to reach the rate SLO, there must be a partition I_1, I_2 of the indexes $1, \ldots, |A|$ such that $\sum_{i \in I_1} w_{t_i} = \sum_{i \in I_2} w_{t_i} = 1$. That is, $\sum_{i \in I_1} \frac{2a_i}{\mathcal{A}} = 1$, or equivalently, $\sum_{i \in I_1} a_i = \frac{\mathcal{A}}{2}$. Deciding whether such a partition exists is equivalent to solving the PARTITION problem. The proof follows. ∎