Programmable Real-time Scheduling of Disaggregated Network Functions

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 that our algorithm can optimize the disaggregated network for real-time processing requirements at the millisecond granularity in just a few control periods.

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 rapid and flexible development of network applications, bringing new real-time industrial applications, such as remote operation, which seemed unfeasible a few years ago, within reach. 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, data plane can use only a small fraction of the time budget. Similar stringent requirements are also imposed by the 5G core network, where the end-to-end latency is 5-10 ms [3]. There is a significant transmission delay (e.g., due to physical distribution) and processing time, which makes only a fraction of time available for the software data plane. On the extreme, an industrial robotic arm motion control leaves a one-way delay between endpoints of 250-1000 µs [2]. It is therefore important for the data plane that the transmission is fast and the traffic of critical applications is real-time.

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. 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. Supported by the ÚNKP-23-4-II-BME-345 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Unfortunately, software VNFs running on commercial off-the-shelf (COTS) hardware usually cannot meet such firm 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.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., [7], [8]). Requiring specialized hardware, however, limits flexibility and feature velocity. Recently there has been substantial work towards defining AI and ML workflows to realize real-time control loops [1], [9]; 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 are still missing* [1]–[3], [10], [11]. We aim to fill this gap in this paper.

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). 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:* 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 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 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.

Numerical Evaluation: We validate our model with measurements executed on the Berkeley Extensible Software Switch (BESS) [12]. Using a custom simulation environment (§V), we demonstrate the effectiveness of our solutions in synthetic

examples and realistic cellular network use cases (§VI).

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 (NFs). The key of our design is a programmable real-time software switch running on COTS hardware, which schedules the execution of the NFs and runs a fast packet processing pipeline as a communication substrate between 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 rate 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 input packet rate and/or delay SLOs.

Answering these questions is not trivial due to the sheer amount of system parameters. On the hardware level, the execution is affected by the CPU and the memory models (e.g., L1-L3 caching, NUMA, memory contention, branch prediction, etc.) [13]. On top of this, the operating system adds additional unknowns: interrupts, the CPU scheduler, 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. 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, and/or the SLOs. We envision real-time scheduling control loop, in which a model-predictive controller adjusts the scheduling weights of the running system in concert with current demands. This model follows state-of-the-art design [1], [14], [15].

A. Dataflow Graphs and Scheduling

Software switches for running disaggregated network functions (FastClick [16], BESS [12], etc.) are usually designed as a *dataflow-graph runtime*. This model is much akin to TensorFlow for ML or GStreamer for multimedia.

In this model, **modules** (or *nodes*) represent packet processing functions (i.e., VNFs). At the most basic level, each module implements a single processing primitive, like parse/deparse, match/action, 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, data plane of a 5G mobile gateway, etc. Our framework is completely agnostic to the choice of modules. 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 form the **dataflow graph**.

VNF 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 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 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) [17]. 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 lower level, 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).

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 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. 1 depicts a sample packet processing pipeline taken from [18] 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 of task_2 is 1 unit, while task_0 modules 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

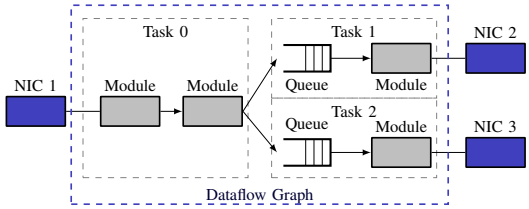


Figure 1. Example Pipeline (see [18]).

$D_1=5$ 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 ${}^{w_1}/_{c_1}={}^1/_4<R_1$ units rate 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. Meanwhile, flow_2 meets its SLOs and there is some slack remaining: it gets ${}^1/_2$ units rate, 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 $2/\frac{4}{5}+2<D_1$ delay, and flow_2 gets ${}^1/_5=R_2$ rate and $5+1=D_2$ delay, both meeting 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 WFQ scheduling weights in order for each flow (i.e., NF chain [19]) to meet its requested rate-SLO and delay-SLO. Our model uses static resource allocation.

Problem 1 (Feasible scheduling): Given a set of static task graphs G_t and a fix 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 $f\in F$.

III. An Optimal Scheduling Algorithm

In the following, given a static resource allocation we define an ideal scheduler for optimal module execution scheduling in a dataflow graph under delay and rate SLOs.

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: $\theta_t=\frac{\sum_{f\in F_t} r_{t,f}\tau_{t,f}}{\sum_{f\in F_t} r_{t,f}}=\frac{\sum_{f\in F_t}(r_{t,f}\sum_{v\in p_{t,f}} c_v)}{\sum_{f\in F_t} r_{t,f}}$. 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 [17], the time between two consecutive runs of a task t equals $\frac{B\theta_t}{T_{i_t} w_t}$ on average.

Proof: We start with the notations of [17], with time quanta and stride scaled to: $\texttt{quanta}=1$, $\texttt{stride}_1=1$. We 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, with time, $\frac{\texttt{Pass}(t)}{\texttt{Pass}(t')}\to 1$, for any $t'\in S_i$ (Eq. infty). With this, the timeshare a task t gets is:

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

Thus, the time accumulates for task t to run once in ${}^{\theta_t}/_{w_t}$. ■

Let us compute the queuing delay first, 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 ${}^Q/_B$ 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 (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: the average packet processing delay for flow f 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 will use this delay estimation in our models; we will 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 our assumptions), 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 $r_{s_f,f} = \rho_f$. W.l.o.g., 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 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 [20]).

D. Feasible WFQ Scheduling Control

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.

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

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

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

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

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

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 to tackle Problem 1.

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 conversation [20], which automatically blocks upstream module execution when some downstream gets overflown (i.e., then 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 to treat the flow rates $r_f = \min_{t:f \in F_t} r_{t,f}$ as constant. This also has the benefit that we no longer need to measure the input rate ρ_f from the running system and now the task processing times $\theta_t = \text{const}$ 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) [14], [15]. Now, $r_{t,f}$ 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 $r_t := \sum_{f \in F_t} r_{t,f}$ to denote the total packet rate of task t.

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:

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

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

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

$$w_t \geq 0 \qquad t \in T. \quad (9)$$

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 [14]. Thus, we will 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:

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

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

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

$$w_t \geq 0 \qquad t \in T \quad (13)$$

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 the 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 $L(w)$.

Proof: Suppose indirectly that there is no optimal solution where the task weights on each worker add up to 1. Take an

Algorithm 1: Projected Gradient Method for a Worker

Input: $\forall t \in S_i$, λ_t and θ_t given, and $w_t[1] = 1/|S_i|$

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 $w[k]$

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. ∎

The final simplified model for answering Problem 1 is as follows, with the objective function denoted by $L(w)$:

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

$$\sum_{t \in S_i} w_t = 1, \ \forall i \in [1, S]; \qquad w_t \geq 0, \ \forall t \in S_i \quad (15)$$

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:

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

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 [21, 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 $\nu_{\max}$ 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:

Algorithm 2: Modified Line Search (Polyline Search)

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

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

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

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

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

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

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

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

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

remove blocked tasks from Φ

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

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

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

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. We claim without proof Prop. 1, showing that our optimization algorithm in each iteration either claims optimality, or departs towards the optimum.

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

V. Simulator

We created a discrete time simulator to experiment with our controllers. The simulator workflow is the following. 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 weights w_t for each task 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:

$$\max \sum_{f \in F} r_f \quad (16)$$

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

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

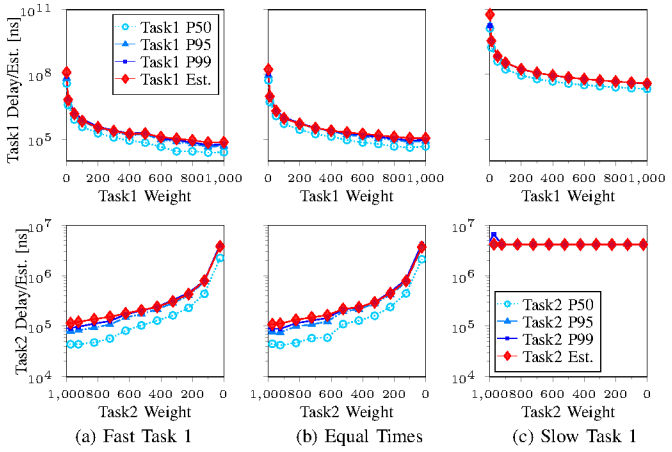
$$r_f = r_{s_f,f} \qquad f \in F \quad (19)$$

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

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

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 the traffic it receives. Based on this, we use the following rule: after solving (16)–(20) set $\lambda_t = 1$ for $t \in T$ 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 (Alg. 1) followed by a single line-search (Alg. 2) to



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

Figure 2. Validating Delay Estimate: Example Pipeline (Fig. 1) 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.

obtain 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 total system time surpasses a given limit. We implemented the simulator in Python [22].

VI. Evaluation

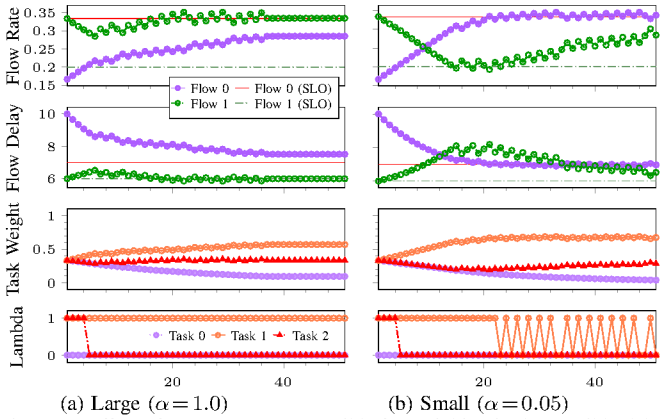
We evaluated our real-time scheduler controller logic in extensive simulation studies. Since the model critically depends on the delay estimate (§III-B), first we confirm this estimate on a real software switch. Then, we present a detailed numerical evaluation of our controller logic (§IV-B) with the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

We will use synthetic and real-life sample pipelines from [15], [20] taken from an official 5G benchmark suite [4] for the evaluations. In particular, for the synthetic tests, we chose the *fork* example pipeline of Fig. 1 and a *taildrop* pipeline consisting of 3 tasks, from which the last one is heavyweight [20]. As a real 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

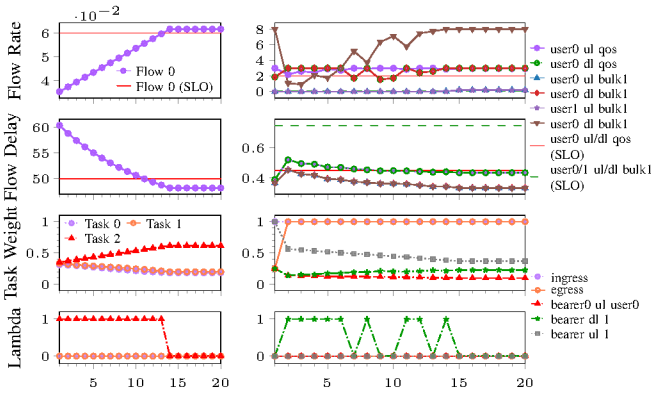
A dependable delay estimation is crucial for scheduling latency-sensitive pipelines. Hence, our evaluation starts with the validation of the task delay estimate of §III-B. For this purpose, we implement the *fork* pipeline (Fig. 1) in a widely-deployed software switch: BESS [12]. We adjust *i)* the fan-out of Task0 (number of tasks connected to the egress module of Task0); *ii)* the execution time of Task1; and *iii)* the ratio of weights between the egress tasks .

Fig. 2 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$^{\text{th}}$ 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.*



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

Figure 3. Effect of parameter α responsible for weighing in the possible delay SLO violations, measured on the *fork* pipeline over 50 iterations.



(a) Taildrop; costs: Task0 & Task1: 1, Task2: 10. (b) MGW; single user on QoS and bulk bearers (uplink/downlink).

Figure 4. Pipelines *taildrop* and *MGW* associated with satisfiable SLOs. With $\alpha = 1$, the controller found a feasible solution in 15 control periods in both cases.

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. 1. 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. 3. 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 real-time applications 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 [23].

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 [20]. Clearly, to remedy this, the scheduling weight of the last task needs to be increased. Fig. 4a 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 taken from the official 5G NFV benchmark suite [4]. A 5G mobile gateway connects mobile user equipments to the Internet. This requires a complex pipeline with differentiated traffic classes (called "bearers"). Traffic flows are either uplink or downlink, and are further classified among bearers. Users may have connections on multiple bearers both in 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 are bulk traffic. The number of concurrent flows is $2\times$ number of users on 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. 4b shows an MGW pipeline with 37 modules organized into 5 tasks and 3 workers: *ingress* and *egress* tasks both have a separate worker, while *bearer dl 1*, *bearer ul 1* and *bearer0 ul user0* tasks share a common worker (full description in our GitHub repo [22]). 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*.

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 [24] or packet I/O [6]. Performance prediction of VNFs is therefore crucial for guaranteeing SLOs. SLOMO [6] predicts collocated VNF performance using ML. Bolt [5] leverages symbolic execution to estimate processing costs of traffic classes processed by a VNF; also generalized to NF chains. As opposed to our work, these methods require extensive profiling. 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 [8], an NFV framework with GPU offload, introduces a multi-layer system with delay prediction model to guarantee delay-SLOs for single VNF deployments. ResQ [24] provides performance isolation at CPU last-level cache solving the noisy neighbor problem of VNFs, and enables enforcing SLOs. Batchy [14], [15] is a dataflow graph scheduler framework, which enables enforcing delay-SLOs. Batchy uses controlled queuing to efficiently reconstruct fragmented batches in accordance with strict SLOs. Contrary to these works, our closed loop scheduler is built on an *analytical system model*, without relying on static performance benchmarks or costly prior ML training. Our controller can handle both rate- and delay-SLOs.

VIII. Conclusions

This paper presents a controller framework for real-time execution of disaggregated services on COTS hardware. The controller relies on a comprehensive analytical model, combining the formal model with monitoring data to rapidly find an optimized schedule. We present a model-based gradient-optimization control algorithm to provide optimal scheduling. Our evaluation results show that the model predicts delays reliably and the control algorithm is able to converge the system to an SLO-compliant state in just a few iterations. Future work involves dynamically adjusting parameters and extending the model to multiple switches.

References

[1] M. Polese *et al.*, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," *Comm. Surveys Tuts.*, p. 1376–1411, 2023.

[2] F. Voigtländer *et al.*, "5G for robotics: Ultra-low latency control of distributed robotic systems," in *IEEE ISCSIC*, 2017.

[3] NGMN Alliance, "5G white paper," *Next generation mobile networks, white paper*, 2015.

[4] T. Lévai *et al.*, "The Price for Programmability in the Software Data Plane: The Vendor Perspective," *IEEE JSAC*, vol. 36, 2018.

[5] R. Iyer *et al.*, "Performance Contracts for Software Network Functions," in *NSDI 19*. Boston, MA: USENIX Association, Feb. 2019, pp. 517–530.

[6] A. Manousis *et al.*, "Contention-Aware Performance Prediction For Virtualized Network Functions," in *ACM SIGCOMM*, 2020.

[7] Z. Zhao *et al.*, "Achieving 100Gbps Intrusion Prevention on a Single Server," in *USENIX OSDI*, 2020, pp. 1083–1100.

[8] Z. Zheng *et al.*, "Grus: Enabling Latency SLOs for GPU-Accelerated NFV Systems," in *IEEE ICNP*, 2018, pp. 154–164.

[9] X. Wang *et al.*, "Self-play learning strategies for resource assignment in Open-RAN networks," *Computer Networks*, vol. 206, p. 108682, 2022.

[10] D. Bankov *et al.*, "Enabling real-time applications in wi-fi networks," *International Journal of Distributed Sensor Networks*, vol. 15, 2019.

[11] W. Azariah *et al.*, "A survey on Open Radio Access Networks: Challenges, research directions, and open source approaches," 2022.

[12] S. Han *et al.*, "SoftNIC: A Software NIC to Augment Hardware," UC Berkeley, Tech. Rep., 2015.

[13] L. Linguaglossa *et al.*, "Survey of performance acceleration techniques for Network Function Virtualization," *Proceedings of the IEEE*, 2019.

[14] T. Lévai *et al.*, "Batch-scheduling Data Flow Graphs with Service-level Objectives on Multicore Systems," *INFOCOMMUNICATIONS JOURNAL*, vol. 14, pp. 43–50, 2022.

[15] T. Lévai *et al.*, "Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives," in *USENIX NSDI 20*, Santa Clara, CA, 2020.

[16] T. Barbette *et al.*, "Fast Userspace Packet Processing," in *ACM/IEEE ANCS*, 2015, pp. 5–16.

[17] C. A. Waldspurger *et al.*, *Stride scheduling: Deterministic proportional share resource management*. MIT, 1995.

[18] C. Lan, "An Architecture for Network Function Virtualization," Ph.D. dissertation, UC Berkeley, 2018.

[19] J. Wang *et al.*, "Quadrant: A Cloud-Deployable NF Virtualization Platform," in *ACM SoCC*, 2022, p. 493–509.

[20] S. G. Kulkarni *et al.*, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," in *ACM SIGCOMM*, 2017, pp. 71–84.

[21] M. S. Bazaraa *et al.*, *Nonlinear programming: theory and algorithms*. John Wiley & Sons, 2013.

[22] "Source code on GitHub," https://github.com/hsnlab/realtime-nf-scheduling.

[23] D. Bansal *et al.*, "Dynamic behavior of slowly-responsive congestion control algorithms," *ACM SIGCOMM CCR*, vol. 31, pp. 263–274, 2001.

[24] A. Tootoonchian *et al.*, "ResQ: Enabling SLOs in Network Function Virtualization," in *USENIX NSDI*, 2018, pp. 283–297.