# Letting off STEAM: Distributed Runtime Traffic Scheduling for Service Function Chaining

Marcel Blöcher
TU Darmstadt
bloecher@dsp.tu-darmstadt.de

Ramin Khalili
Huawei Technologies, Munich
ramin.khalili@huawei.com

Lin Wang
VU Amsterdam
& TU Darmstadt
lin.wang@vu.nl

Patrick Eugster
Università della Svizzera italiana (USI)
& Purdue University & TU Darmstadt
eugstp@usi.ch

***Abstract*—Network function virtualization has introduced a high degree of flexibility for orchestrating service functions. The provisioning of chains of service functions requires making decisions on both (1) placement of service functions and (2) scheduling of traffic through them. The placement problem (1) can be tackled during the planning phase, by exploiting coarse-grained traffic information, and has been studied extensively. However, *runtime* traffic scheduling (2) for optimizing system utilization and service quality, as required for future edge cloud and mobile carrier scenarios, has not been addressed so far.**

**We fill this gap by presenting a queuing-based system model to characterize the *runtime traffic scheduling problem* for service function chaining. We propose a throughput-optimal scheduling policy, called integer allocation maximum pressure policy (IA-MPP). To ensure practicality in large distributed settings, we propose multi-site cooperative IA-MPP (STEAM), fulfilling runtime requirements while achieving near-optimal performance. We examine our policies in various settings representing real-world scenarios. STEAM closely matches IA-MPP in terms of throughput, and significantly outperforms (possible adaptations of) existing static or coarse-grained dynamic solutions, requiring $30\%$-$60\%$ less server capacity for similar service quality. Our STEAM prototype shows feasibility running on a standard server.**

***Index Terms*—Service function chaining, runtime traffic scheduling, stochastic processing networks**

## I. Introduction

Advances in core technologies like network function virtualization (NFV) have laid the foundation for envisioning a dynamic multi-service network architecture for 5G and beyond, where communication spans data centers, carrier networks, and edge locations, accessing dynamically created services with computing and storage resources distributed throughout the network. NSF's 18-535 [1] and EU's ICT-20 [2] calls for proposals detail the demand for providing such end-to-end network services, which steer packets through sequences of service function instances (SFIs), referred to as service function chains (SFCs) [3]. However, one missing component towards these visions, especially considering that service functions (SFs) increasingly include application-specific services [4]–[6], are SFCs provisioning mechanisms [1], [7], [8], that are capable of creating SFC instances, and scheduling traffic through them, *on the fly* [1]. This requires making decision on both **(1) placement** of SFIs and **(2) scheduling of traffic** through them [9].

Most existing works focus on the placement problem (1), deciding *where* SFIs should be deployed (e.g., on which server) and *how many* resources (e.g., CPU shares) should be assigned to each of them [10]–[15]. These solutions perform chaining of SFs in a mostly *static* manner, where traffic is steered through deployed SFIs in the network with load-balancing performed among them. Few *dynamic* solutions are discussed [7], [8], [16]–[19], in which the deployment of these SFIs and their resource assignments are periodically adapted to changes in network traffic and topology, as required for future carrier networks. However, these solutions are still coarse-grained [17], where the adaptation takes seconds to take effect [19], or cannot be applied in real time due to its high complexity [7] and the involvement of disruptive SFIs migration [17]. Therefore, these proposals are not able to explore and exploit the resources that become available on the fly as a result of real-time, sudden, changes in network traffic. Yet, as we shall show, such a fine-grained approach based on per-packet scheduling is required for achieving high resource utilization under high traffic dynamics.

Differently from previous work, in this paper, we thus treat the SFC traffic scheduling problem (2) as a **runtime scheduling problem**. The goal is to dynamically assign packets with specific processing requests, to active SFIs in the network. We assume that SFIs are already deployed on servers using any of the algorithms for (1). However, these SFIs are not pre-assigned any resources or traffic. Our goal is to select an appropriate SFI *for each packet* and to decide on the amount of resources that should be assigned to each SFI at runtime. The major challenge is to quickly react to dynamic traffic conditions, without any *a priori* knowledge of traffic distribution.

We characterize the SFC traffic scheduling problem with a stochastic model and show that this problem can be reduced to the scheduling problem in a stochastic processing network (SPN) [20]. We propose the **integer allocation maximum pressure policy (IA-MPP)** for SFC scheduling, a derivation of maximum pressure policy (MPP), which we show is throughput-optimal. It is also asymptotically optimal for minimizing a cost function of buffer occupancy levels in the network, providing approximate guarantees on latency. Furthermore, we show that

This work has been co-funded by the Federal Ministry of Education and Research (BMBF) Software Campus grant 01IS17050 "dynSFC", the German Research Foundation (DFG) as part of the projects B2 and C7 in the Collaborative Research Center (CRC) 1053 "MAKI" and DFG grant 392046569 (61761136014 for NSFC), and the EU H2020 program under grant ICT-815279 "5G-VINNI" and ERC grant FP7-617805 "LiveSoft". We would like to thank Daniel Failing for his support building the STEAM prototype.

TABLE I: Notation used

| Symbol | Description |
|---|---|
| $\mathcal{V}$ | Set of SFFs in the network |
| $\bar{d}_E$ | Average network delay of link $E \in \mathcal{E}$ |
| $\mathcal{S}$ | Set of servers |
| $c_S$ | Processing capacity of server $S \in \mathcal{S}$ |
| $\mathcal{F}$ | Set of SFs |
| $\mu_F$ | Processing rate of SF $F$ using one resource unit |
| $\mathcal{I}$, $\mathcal{I}_S$ | Set of all SFIs and of those running on server $S$ |
| $w_I$ | Resource share of SFI $I$ at a server |
| $\mathcal{B}$ | Set of all buffers at all SFFs |
| $\vec{z}$ | Vector of buffer utilization levels |
| $\mathcal{A}$ | Set of all activities for the corresponding SPN |
| $\mathcal{H}(t)$ | Set of all feasible allocations at time $t$ |
| $\mathbf{R}$ | Input-output matrix of the network |
| $\theta_l$, $\theta_h$ | Thresholds used by SALVE |
| $\phi_b$, $\phi_{w,S}$ | Batch size and threshold of $s \in \mathcal{S}$ used by STEAM |

the time complexity of IA-MPP is bounded by a linear term on the number of sites in the network. Importantly, IA-MPP requires no *a priori* information about network traffic patterns.

Based on practical constraints in large deployments, we present a novel distributed variant of our solution dubbed **multi-site cooperative IA-MPP (STEAM)**, where a scheduler instance is running at each site *using only site-local state*, and is invoked for batches of packets. We study the performance of STEAM using a packet-level simulator as well as a prototype implementation based on DPDK [21]. We observe that STEAM performs closely to the optimum (IA-MPP) and significantly outperforms (possible adaptation of) existing static or coarse-grained dynamic solutions. Specifically, STEAM improves resource usage, requiring much fewer resources to achieve similar service quality. The prototype implementation of STEAM shows the feasibility of our runtime solution.

The main contributions of this paper are as follows:

1) We introduce a model of SFC provisioning infrastructure based on the SFC RFC [9], and formulate the SFC runtime scheduling problem. We show that our problem can be reduced to the scheduling problem in an SPN.
2) We present a throughput-optimal solution IA-MPP, with linear time complexity (in the number of sites), where schedulers have access to each other's state.
3) We introduce a distributed heuristic STEAM, where schedulers have access only to their local state and scheduling costs are amortized over batches.
4) We evaluate our solutions based on a discrete-event packet-level simulator, showing that our solutions significantly outperform dynamic variants of existing solutions: (i) STEAM reduces the required amount of resources by 30%-60% compared to the baselines, while providing similar or even better service quality; (ii) STEAM's scheduling quality does not suffer from small batch sizes ($\leq$ 64), making runtime scheduling feasible in practice.
5) We describe a prototype implementation of STEAM and show the feasibility of running STEAM in real-time, achieving 1-4 $10^6/s$ scheduling decisions (1 CPU core).

We introduce our SFC model in § II. § III presents IA-MPP and § IV introduces STEAM. § V evaluates our solutions, § VI

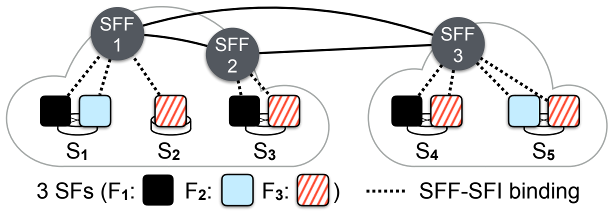


Fig. 1: Small scenario with two sites, three SFFs, five servers, and three SFs with multiple SFIs of each.

discusses related work, and § VII draws conclusions.

## II. Model and Problem

In this section we introduce a comprehensive model for the runtime traffic scheduling problem for SFC. We use calligraphic fonts for sets (e.g., $\mathcal{S}$), capital letters to refer to members of a set (e.g., $S \in \mathcal{S}$), lower-case letters to refer to variables (e.g., $v$), and letters with arrows, such as $\vec{z}$, to refer to vectors. For two vectors $\vec{x}$, $\vec{y}$ of the same size, $\vec{x} \times \vec{y}$ denotes the cross product of the vectors and $\vec{x} \cdot \vec{y}$ denotes their dot product. Table I summarizes major notation used.

### A. System Model

*1) Infrastructure.* We consider an architecture similar to the one proposed in RFC 7665 [9]. Our network consists of geographically distributed sites, each of which holds servers for running SFIs as depicted in Fig. 1. Attached to each site is a set of service function forwarders (SFFs), which are responsible for forwarding traffic within their site and among sites. We model the network of SFFs across sites with a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ the set of SFFs and $\mathcal{E}$ the set of links interconnecting SFFs. For any link $E \in \mathcal{E}$, $d_E(l)$ denotes the $l$-th packet propagation latency, where $\{d_E(l), l \geq 1\}$ is assumed to be a sequence of i.i.d. random variables, with an average noted $\bar{d}_E$ and a finite variance. We thus consider that this propagation latency is time-varying. Moreover, we assume that network planning, as discussed in [22], [23] for different use cases, is performed beforehand, as a result of which enough capacity is assigned to transmission links among SFFs.

Attached to each SFF, we have a set of servers each running SFIs. $\mathcal{S}$ denotes the set of all servers in the system. Each server $S \in \mathcal{S}$ has a total processing capacity of $c_S$ ($c_S > 0$), and hosts at least one SFI. In addition, each SFI belongs to one of the SFFs at the same site, which means that this respective SFF considers the SFI for scheduling purposes. Servers and SFFs in a site are interconnected by a high-throughput low-latency network. For the sake of tractability, our optimal solution (§ III) assumes no latency and bandwidth constraints for communication *within* a site. This assumption is relatively realistic as modern data center networks can provide ultra-low latency and full bisection bandwidth between any pair of servers [24], [25]. We relax the assumptions in § IV.

*2) SFIs.* An SF is a piece (type) of processing logic applied to network packets, while an SFI is an instantiation of an SF deployed on a server. For simplicity, we focus on stateless SFIs, where packets from the same flow can be scheduled separately. (Stateful SFIs can be built on top of stateless SFIs using a distributed data layer [26], [27].) We denote by $\mathcal{F}$ the set of all SFs in the system. Multiple SFIs of the same SF $F \in \mathcal{F}$ might be deployed in the network (see Fig. 1 as an example). We denote by $\mathcal{I}$ the set of all running SFIs of all SFs in the network and by $\mathcal{I}_S$ the set of SFIs running on server $S$.

We consider that SFIs are already deployed in the network, by using any of the solutions proposed in the literature (e.g., [10], [12]; see also § VI). Differently from these studies, however, the SFIs are not pre-assigned any resources or traffic. The scheduler dynamically decides where to send a packet and how many resources to assign to each SFI. Without loss of generality [28]–[31], we assume that all SFIs of the same type in the system have the same processing rate when given equal resources. We denote by $\mu_F$ the processing rate of SF $F \in \mathcal{F}$ when provided one resource unit. Thus, $k \cdot \mu_F$ is the processing rate of an SFI of type $F$ using $k$ units of resources. Moreover, we assume that the processing capacity of a server is shared among all co-located SFIs according to some given policy. Under such a policy, an SFI $I \in \mathcal{I}_S$ receives a share of $w_I$ of the total capacity of server $S \in \mathcal{S}$ and the total capacity of the server is constrained by enforcing $\sum_{I \in \mathcal{I}_S} w_I \leq 1$. Furthermore, we assume that each SFI holds a local buffer to store incoming packets and applies non-preemptive execution of the corresponding SF. The assumptions indicated above are not restrictive, representing many real-world use cases [32].

*3) Network traffic.* Each SFF $V \in \mathcal{V}$ runs a classifier and a scheduler and also behaves as both ingress and egress for the network traffic. We assume that the network traffic is composed of many flows originating from different users connected to the network. The set of ingress and egress nodes of a flow, which are SFFs in $\mathcal{V}$, is determined using the source and destination addresses of the flow. We assume that there are buffers at each SFF, which store incoming packets (new packets arriving to the site via this SFF, or packets forwarded by other SFFs) before scheduling over the available resources.

Like in RFC 7665 [9], we assume that the *classification* of packets in the network is known. This classification is performed at the ingress node of the flow and the classification information can be embedded in the header of each packet of the flow, e.g., by using network service headers (NSHs) [33]. Each packet, after classification, will be assigned an SFC that the packet has to go through. The packet header maintains the processing stage of the packet, specifying by which SF in its SFC it is to be processed next. An SFC in the system is specified by an ordered set of SFs that a flow packet should be processed through, i.e., $C = (F_1, ..., F_k)$, $F_1, ..., F_k \in \mathcal{F}$, where $k$ is the number of SFs on the SFC. In addition, each SFC $C$ is given a set of quality of service (QoS) metrics that the handling of packets undergoing $C$ has to conform to, which in our considered scenarios contains the end-to-end delay. $(F_1, F_2) \in C$ denotes that both $F_1$ and $F_2$ are part of $C$ and that $F_1$ precedes $F_2$ in $C$. An SF in $C$ can be handled by any of its corresponding SFIs deployed in the network.

### B. Problem Description

We now describe the **SFC runtime traffic scheduling problem** (in short, SFC scheduling problem). A *packet class* defines a set of packets in the network (1) to which the same SFC needs to be applied and (2) which are at the same processing stage within that SFC. At each SFF, we maintain a set of buffers, each holding packets falling into a same packet class. The SFC scheduling problem consists in deciding at runtime how to assign packets from buffers to the corresponding SFIs and how to allocate the resources to SFIs. For each packet, the end-to-end delay is the sum of the delays at SFFs, SFIs, and propagation delays between SFFs. Our objective is to maximize the system's processing throughput, while constraining the average delay experienced by packets.

## III. Optimal Scheduling Policy

We show that the SFC scheduling problem is reducible to the scheduling problem in stochastic processing networks (SPNs) [28], and propose a scheduling policy achieving the above objectives (§ II-B) with SFFs accessing each other's state.

### A. Background on SPNs

SPNs are a general class of network models that have been used to characterize a wide range of application fields [20], including manufacturing systems and cross-training of workers at a call center. The key elements of an SPN include a set of *buffers*, a set of *processors*, and a set of *activities*. Each buffer holds jobs that await service. Each activity takes job(s) from at least one of the buffers and requires at least one processor available to process the job(s). A job departing after service from a buffer will next be routed to another buffer, or leave the network, with probabilities depending on the activity taken.

### B. Reducing SFC Scheduling to SPN Scheduling

With an ideal setting, the SFC scheduling problem can be reduced to a variant of the scheduling problem in an SPN.

*1) Buffer.* According to our system model, each SFF in the network holds a number of buffers which are used to store packets. All incoming packets of the same packet class at an SFF are stored in the same buffer. We denote by $\mathcal{B}$ the total set of buffers in the system. A packet's class can be determined at an SFF by extracting information encapsulated in the packet header. When an SFF receives a packet, it determines the packet's class and pushes it into the corresponding buffer. Packets in the same buffer are processed in FIFO order.

*2) Processor.* Each server in our model corresponds to a processor in an SPN. Each server can process packets belonging to the packet classes handled by its SFIs, regardless of its location. As there can be multiple SFIs for the same SF, multiple servers can process packets from the same buffer.

*3) Activity.* We define an activity as the processing of a packet from a buffer $B \in \mathcal{B}$ by an eligible server, i.e., a server in $\mathcal{S}$ which contains an SFI of the required SF. The total set of activities can be expressed by $\mathcal{A} = \{B \mapsto S \mid B \in \mathcal{B} \wedge S \in \mathcal{S}_B\}$, where $\mathcal{S}_B \subseteq \mathcal{S}$ is the set of eligible servers for packets in buffer $B$. $B \mapsto S$ denotes an activity which processes packets from a buffer $B$ over a server $S$. We denote by $\mathcal{A}_B$ the set of activities connected to buffer $B$ and by $\mathcal{A}_S$ the set of activities connected to server $S$. Associated with each activity $A \in \mathcal{A}$ is a processing rate $\mu_A$ that determines the rate at which a packet will be processed by this activity. The processing rate depends on the relation of the server, buffer, and SFF. If the server and the buffer are under the control of the same SFF, the processing rate is given by the service rate of the corresponding SFI, i.e., $\mu_A = \mu_F$ where $F$ is the SF of the SFI; otherwise, the processing rate of the activity is given by a function $g(\cdot)$ of the latency between the corresponding SFFs and the SFI's service rate, i.e., $\mu_A = g(\mu_F, \bar{d}_E)$ where $E$ is the link between the SFF holding the buffer and the SFF controlling the SFI.

*4) Routing.* Each packet from a buffer $B$, once being served by an activity $A$, changes its packet class and gets injected into buffer $B'$ or leaves the network. We define by $p^A_{BB'}$ the probability that a packet from buffer $B$ is injected into buffer $B'$. Consequently, $1 - \sum_{B' \in \mathcal{B}} p^A_{BB'}$ is the probability that the packet leaves the network. The packet's class transitions as its processing stage is advanced by one SF after being served by the activity. In our model, $p^A_{BB'}$ has a very simple form. If a buffer $B$ holds packets at the last stage within their SFC, then $p^A_{BB'} = 0$ for all $B' \in \mathcal{B}$. For any other $B \in \mathcal{B}$, there always exists one $B' \in \mathcal{B}$ such that $p^A_{BB'} = 1$, else $p^A_{BB'} = 0$.

*5) Example.* To further clarify the above mappings, we take the example in Fig. 1 and consider two SFCs: $C_1 = (F_1, F_2, F_3)$ and $C_2 = (F_2)$. We have four packet classes: (1) packets with SFC $C_1$ at their first processing stage (to be processed by $F_1$); (2) packets with SFC $C_1$ at their second stage (to be processed by $F_2$); (3) packets with SFC $C_1$ at their last stage (to be processed by $F_3$); (4) packets with SFC $C_2$ to be processed by $F_2$. As we have 3 SFFs, we would have a total of 12 buffers as depicted in Fig. 2. Activities $A_1$, $A_2$, and $A_3$ connect buffer $B_1$ to $S_1$, $S_3$, and $S_4$, respectively. $\mu_{A_1}$, the processing rate of $A_1$, is $\mu_{F_1}$ – the processing rate of SFI of type $F_1$; $\mu_{A_2} = g(\mu_{F_1}, \bar{d}_{E_1})$, where $E_1$ is the link between SFF1 and SFF2; and $\mu_{A_3} = g(\mu_{F_1}, \bar{d}_{E_2})$, with $E_2$ being the link between SFF1 and SFF3. When a packet in buffer $B_1$ is served by any of the connected activities, it changes its class and is injected into the corresponding next buffer – $B_2$ if it is served by $A_1$, $B_6$ if served by $A_2$, $B_{10}$ if served by $A_3$ etc.

*6) Reduction to SPN.* Knowing the topology of $\mathcal{G}$, the set of servers $\mathcal{S}$, the set of SFCs, and the set of SFIs $\mathcal{I}$, we can determine $\mathcal{B}$, $\mathcal{S}$, $\mathcal{A}$, and $p^A_{BB'}$. In the ideal case, we assume that at any time $t$, all schedulers in the network are aware of the state of all buffers, i.e., the buffer utilization level which is given by $\vec{z}(t)$, a vector of size $|\mathcal{B}|$, and also of the state of every $S \in \mathcal{S}$, $q_S(t) = \{0, 1\}$, where $q_S(t) = 0$ if $S$ is idle, else 1. Our SFC scheduling problem is then reducible to the SPN scheduling problem [30], aiming at designing a control policy for the activities such that the SPN's throughput is maximized, while ensuring that all the buffers are stabilized.

## C. Integer Allocation Maximum Pressure Policy (IA-MPP)

Dai and Lin [30] show that the optimal scheduling can be obtained for SPNs by following the maximum pressure policy

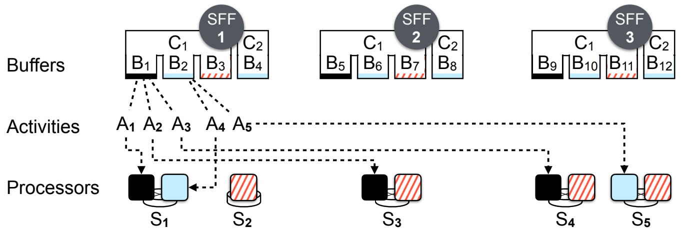


Fig. 2: An SPN representation of the scenario in Fig. 1 with two SFCs $C_1 = (F_1, F_2, F_3)$, $C_2 = (F_2)$. Showing only $A_1$-$A_5$.

(MPP). We prove that a simplified version of MPP, IA-MPP can be applied to the SFC scheduling problem. IA-MPP also achieves optimality, but with much less computation than MPP.

The essential decision we have to make immediately is on the amount of resources allocated to each of the activities at a server when it becomes idle. We denote by a vector $\vec{h}$ of size $|\mathcal{A}|$ an allocation. A feasible allocation has to satisfy $0 \le h_A \le 1, A \in \mathcal{A}$. If an activity performs at level $h_A$, it consumes a fraction of $h_A$ resources of the corresponding server. Note that $\sum_{A \in \mathcal{A}_S} h_A = 1, \forall S \in \mathcal{S}$. Let $\mathcal{H}(t)$ be the set of all feasible allocations in the network at time $t$. For each buffer $B \in \mathcal{B}$ and each activity $A \in \mathcal{A}$, we define

$$r_{BA} = \begin{cases} \mu_A & A \in \mathcal{A}_B, \\ -\mu_A & A \in \mathcal{A}_{B'} \text{ and } p^A_{B'B} = 1, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

$\mathbf{R} = (r_{BA})$ is called the *input-output matrix* of the network. It captures the average processing rates of packets from buffer $B$ consumed by activity $A$, as introduced in [28]. Given a weight vector $\vec{\alpha}$ of size $|\mathcal{B}|$, we define by $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t)) = (\vec{\alpha} \times \vec{z}(t)) \cdot \mathbf{R}\vec{h}$ the *network pressure* at time $t$ with parameter $\vec{\alpha}$ under allocation $\vec{h} \in \mathcal{H}(t)$ and buffer utilization level $\vec{z}(t)$. MPP aims to maximize network pressure by picking suitable allocations:

$$\vec{h}^* \in \arg\max_{\vec{h} \in \mathcal{H}(t)} \Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t)). \tag{2}$$

Note that $\mathcal{H}(t)$ is bounded and convex. As $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t))$ is linear in $\vec{h}$, the maximum of $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}(t))$ will be achieved at one of the extreme points. We can prove that the existence of an extreme allocation for maximum network pressure is ensured.

**Lemma 1.** *For any buffer level $\vec{z}(t)$ ($z_B(t) \ge 0, \forall B \in \mathcal{B}$), there exists an extreme allocation $\vec{h}^* \in \mathcal{H}(t)$ that maximizes the network pressure $\Phi(\vec{h}, \vec{z}(t))$ such that for each constituent buffer $B$ of $\vec{h}^*$, the buffer level $z_B(t)$ is positive.*

*Proof.* The network we consider is strict Leontief [34] as each activity is associated with exactly one buffer. The lemma follows directly if we consider preemptive scheduling [30]. With non-preemption, the lemma holds when the network is reversed Leontief. This is (also) the case here as in our model each activity needs exactly one processor to be active. □

**Lemma 2.** *The extreme allocation $\vec{h}^*$ for maximum network pressure is an integer allocation.*

*Proof.* An allocation $\mathcal{A}$ is called an integer allocation if it satisfies $h_A \in \{0,1\}, \forall A \in \mathcal{A}$. We assume that when the processor is idle, it takes on a dummy activity $A_0$. Thus, processor $S$ will be able to take any of the activities in $\mathcal{A}_S^0 = A_0 \cup \mathcal{A}_S$. We now prove the lemma by contradiction. Suppose we are given an extreme allocation $\vec{h}$ where $\exists \tilde{A} \in \mathcal{A}$ such that $h_{\tilde{A}} \in (0,1)$. Let $\tilde{S}$ be the processor that holds activity $\tilde{A}$. (Note that activity $\tilde{A}$ requires only one processor due to the fact that our network is reversed Leontief.) For each $A \in \mathcal{A}_{\tilde{S}}^0$, we define a new allocation $\vec{h}'(A)$ by modifying $\vec{h}$ in the following way: We process $A$ with $h_A = 1$ at processor $\tilde{S}$ and keep the allocation on other servers unchanged. It is easy to check that $\vec{h}'(A)$ is a feasible allocation. It follows that $\vec{h} = \sum_{A \in \mathcal{A}_{\tilde{S}}^0} h_A \vec{h}'(A)$, where we set $h_{A_0} = 1 - \sum_{A \in \mathcal{A}_{\tilde{S}}^0, A \neq A_0} h_A$. Since $\sum_{A \in \mathcal{A}_{\tilde{S}}^0} h_A = 1$, $\tilde{A} \in \mathcal{A}_{\tilde{S}}^0$, and $h_{\tilde{A}} < 1$, the summation contains at least two terms. As a result, $\vec{h}$ is a linear combination of at least two feasible allocations and thus, it cannot be an extreme allocation, contradicting the assumption. Hence any extreme allocation must be integer. □

This shows that the allocation produced by MPP in our SFC scheduling problem never splits the processing capacity of a processor. We thus refer to this version of MPP as **IA-MPP**. This property gives us the following network stability result.

**Theorem 1.** *The network operating under a non-preemptive IA-MPP can be stabilized if ever possible.*

*Proof.* To prove this, we first introduce an auxiliary linear program called *static planning problem* defined by Harrison [28]:

$$\min \rho \text{ s.t. } \mathbf{R}\vec{x} = 0; \sum_{A \in \mathcal{A}_S} x_A \leq \rho, \forall S \in \mathcal{S}; x_A > 0, \forall A \in \mathcal{A}.$$

Here $\vec{x}$ is a column vector of size $|\mathcal{A}|$ representing the long-run fraction of time during which each activity is used. The above problem indicates that the long-run input rate to the buffer is equal to the long-run output rate from the buffer. According to Theorem 1 proposed in [29], the static planning problem has a feasible solution with $\rho \leq 1$ if the network is stable under some service policy. On the other hand, applying Theorem 9 of the same work [29], we can prove that the non-preemptive non-processor-splitting IA-MPP can stabilize the network if the static planning problem has a feasible solution with $\rho \leq 1$ considering the fact that our network is reversed Leontief. □

**Corollary 1.** *For any $\vec{\alpha} > 0$, IA-MPP with parameter $\vec{\alpha}$ is asymptotically optimal with respect to network throughput.*

*Proof.* Lemma 1 implies that our network model and assumptions satisfy the extreme-allocation-available (EAA) condition. Combined with Theorem 1, IA-MPP with parameter $\vec{\alpha}$ is asymptotically efficient according to Theorem 1 in [30]. □

**Theorem 2.** *For any given $\varepsilon > 0$, there exists an IA-MPP $\vec{h}^*$ that is asymptotically optimal for a quadratic cost function of the buffer level $\vec{z}(t)$, i.e., $\sum_{B \in \mathcal{B}} \alpha_B (z_B(t))^2$.*

The proof of the above theorem follows from the fact that our network model and assumptions satisfy Assumptions 1-4 in [30]. Thus, the same result on asymptotic optimality of quadratic holding cost in Theorem 3 from [30] applies here. This result basically provides a theoretical estimation of the buffer level and thus, implies a rough guarantee on network latency since queuing latency is usually the dominant factor during the entire packet processing. We will further validate end-to-end latency for packet processing in the network in § V.

Following Lemma 2, IA-MPP can be simplified as follows. For any $S \in \mathcal{S}$, and any activity $A \in \mathcal{A}_S$, we define

$$\Phi_{AS} = \textstyle\sum_{B \in \mathcal{B}} \alpha_B r_{BA} z_B(t). \tag{3}$$

If processor $S$ is in idle state at time $t$, the scheduler selects

$$A^* \in \arg\max_{A \in \mathcal{A}_S} \Phi_{AS} \tag{4}$$

to be served over the server. When more than one allocation attains the maximum, a tie-breaking rule will be applied. Note that the solution space for Eq. 4 is much smaller than that for Eq. 2, requiring much less computation as a consequence.

**Lemma 3.** *The IA-MPP scheduler has a time complexity of $O(|\mathcal{V}|)$, with $|\mathcal{V}|$ the total number of SFFs in the network.*

*Proof.* To find optimal allocation, and for a given $S \in \mathcal{S}$, we need to perform the calculation in Eq. 3 for all $A \in \mathcal{A}_S$ and then apply Eq. 4. Note that $r_{BA}$ under the summation has nonzero values for only one or two $B \in \mathcal{B}$ (refer to Eq. 1). The calculation in Eq. 3 can be reduced to summation of two terms, and hence has $O(1)$ complexity. The IA-MPP calculation thus has complexity of $O(|\mathcal{B}|)$ as $|\mathcal{A}_S| \leq |\mathcal{B}|$. Furthermore, we have $|\mathcal{B}| = k|\mathcal{V}|$ where $k$ is the total number of packet classes which is a constant for a given network. □

This lemma indicates that time complexity of IA-MPP scales with the number of SFFs, which we expect to be much smaller than the number of servers or the number of SFIs.

## IV. Distributed Scheduling Policy

While scheduling optimally, IA-MPP assumes that schedulers can access each other's state. This can become problematic in distributed, multi-site, setups, when such accesses cannot be synchronized instantly. In this section we thus propose a distributed variant of IA-MPP which takes into account the constraints of a deployable scheduler, disabling cross-scheduler accesses and considering link latencies for scheduling.

### A. STEAM Overview

We now propose multi-$\underline{\text{s}}$i$\underline{\text{t}}$e coop$\underline{\text{e}}$r$\underline{\text{a}}$tive IA-$\underline{\text{M}}$PP (STEAM), which is an adaptation of IA-MPP to a distributed setting. In short, with STEAM, each SFF runs its own scheduler using only site-local state, together with an admission control policy (ACP) module. Furthermore scheduling is performed on batches.

*1) Local state.* We consider a multi-site setting where a scheduler instance running at SFF $V \in \mathcal{V}$ has only site-local information: the state of (1) buffers at $V$ (e.g., buffer occupancy levels), (2) SFIs of $V$ (e.g., workload), and (3) servers running these SFIs (e.g., busy or idle). Topological information (e.g., where SFIs of other SFFs are running) is static and thus pertains to global information known to all scheduler instances.

*2) Admission control policy.* For the distributed scheduling problem, an SFF decides whether to serve a packet by an own local SFI, or by a remote SFI. This decision is performed by an ACP module called **<u>S</u>TEAM T-v<u>alve</u> (SALVE)**. If no SFI of the required SF is available locally, the packet must be forwarded to another SFF. SALVE balances load among SFFs by forwarding packets when local traffic load is too high.

To measure traffic load, SALVE estimates the arrival rates and the service rates for each SF the SFF has SFIs for, using an exponentially weighted moving average estimator with a fixed history length, taking also into account traffic bursts [35]. Using these estimations, SALVE applies a threshold-based mechanism to decide whether to serve a packet locally or by other SFFs. Specifically, we use a pair of thresholds $\theta_l \leq \theta_h$. We define the traffic load $tl$ as the ratio of the rate estimator of the packets arriving, and the rate estimator of the corresponding service rate. For each incoming packet, SALVE checks the $tl$ of the SF related to the packet's next step and performs the following: if $tl < \theta_l$, the packet is processed locally; if $\theta_l \leq tl \leq \theta_h$, it is processed locally with probability $1 - \frac{load - \theta_l}{\theta_h - \theta_l}$ and forwarded to other SFFs otherwise; if $tl > \theta_h$, it is forwarded to other SFFs. Note that SALVE updates its arrival rate estimation only when handing off a packet to STEAM. To prevent forwarding loops, SALVE keeps track of each packet's detour count, and drops a packet if this number is above a threshold. We thus use the TTL header of NSH, which is intended for loop detection of SFCs and comes at no additional cost [33].

When SALVE decides to detour a packet, it applies a weighted round-robin mechanism to choose among all SFFs which have at least one matching SFI to serve this packet. We use the total processing capacity of each SFF's servers (with matching SFIs) to set the weights. Note that server capacities are static, hence SALVE calculates the weights offline.

*3) Scheduler.* STEAM takes the scheduling logic from IA-MPP, but considers the network to consist only of the buffers at the local SFF, local server state, and the activities assigning these buffers to these servers. Whenever a local server is idle, STEAM decides the next activity using a modified Eq. 3:

$\Phi_{AS} = \sum_{B \in \mathcal{B}} \alpha_B \hat{r}_{BA} \hat{z}_B(t)$. Here $\vec{\hat{z}}(t)$ is the local buffer utilization level and $\hat{\mathbf{R}} = (\hat{r}_{BA})$ is the local input-output matrix, with values $\vec{z}(t)$ and $\mathbf{R}$ for buffers and activities that are local and zero otherwise.

*4) Batch scheduling.* IA-MPP schedules a packet over a server when the server is idle. However, per-packet runtime scheduling may not fit well with large deployable systems mainly for two reasons: (1) Per-packet runtime scheduling introduces a runtime overhead for each packet, resulting in high system load at the SFF even if the scheduling logic is lightweight. (2) Taking a server into account for scheduling only if the server is idle is optimal in theory when link delays are negligible compared to processing delays at the servers. This might however not always be the case in practice.

STEAM thus uses a packet threshold $\phi_{w,S}$ for each of its servers and applies batch scheduling with batch size $\phi_b$. The batch size $\phi_b$ specifies the (maximum) number of packets STEAM sends over to a server at each scheduling round. More precisely, STEAM uses for each of its servers $S \in \mathcal{S}$ a threshold $\phi_{w,S}$ equaling the number of packets the fastest SFI of server $S$ is able to process within the expected round-trip time (RTT) between the server and the SFF. If there are less than $\phi_{w,S}$ packets on the way or queued at a server, STEAM considers this server to be available for taking a scheduling decision, sending up to $\phi_b$ packets from the selected buffer to this server.

Using $\phi_b$ and $\phi_{w,S}$ reduces the scheduling granularity to one decision per batch and also reduce the effect of link delays. However, the larger the batches, the fewer possibilities STEAM has for choosing the "best" scheduling decision. § V-E investigates the effects of choosing $\phi_b$.

Note that with IA-MPP there is no need for a separate resource sharing policy at the server since the share each SFI receives is inherently dictated by the scheduling decision. When batch scheduling is enabled, we employ a round-robin policy at each server. Since $\phi_{w,S}$ and $\phi_b$ are very small in general, the impact of such a round-robin policy is considered negligible.

## *B. STEAM Deployment*

While focusing on the theoretical design and concepts of STEAM in this paper, we consider practical constraints of an implementation as well. Following Eiffel [36], which shows feasibility of software packet schedulers running at high packet rates, we implement our STEAM prototype as a software scheduler and show its feasibility in § V-E. Besides using a software scheduler, we consider white-box switches [37] and servers with SmartNICs [38] as possible deployment targets.

## V. Evaluation

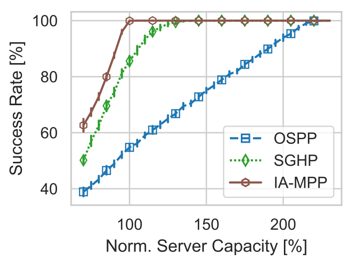
We conducted performance evaluation with large-scale simulations as well as a prototype implementation. Our packet-level discrete event simulator simulates scenarios in compliance with RFC 7665 [9], comprising the network topology including link latencies, packet handling at SFFs, SFIs, and servers, the processing of the SFIs running on servers, and the schedulers.

## *A. Algorithms Compared Against*

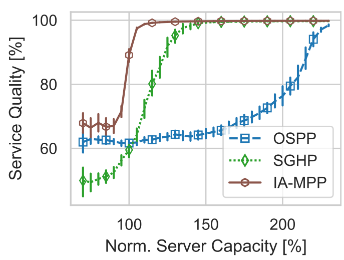
We compare IA-MPP and STEAM with the following two variants of existing static or coarse-grained dynamic algorithms.

*1)* ***OSPP****.* As a variant of [8], [39], the offline static planning policy (OSPP) performs offline planning ahead of traffic arrival, but applies runtime load balancing to react to sudden traffic changes. Similarly to these solutions, if multiple SFIs of the same SF are available, OSPP distributes the traffic using service rate of SFIs as weights, while also considering latency between SFFs – favoring higher-capacity SFIs closer to a packet's egress.

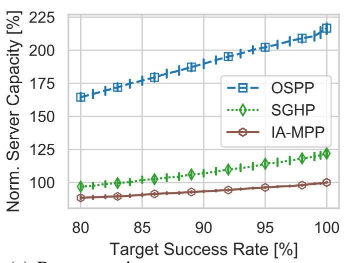
*2)* ***SGHP****.* The second scheduler, shortened greedy heuristic policy (SGHP), adapts the most recent existing heuristics SGH [15] and SPH [7] which do not require any *a priori* information like arrival rate or resource demand of a request. Upon receiving a packet, SGHP extends the routing path iteratively and selects the next SFI among all possible *site-local* ones which is likely to provide the shortest delay to serve the packet based on link latency and queue state information. If the load of the local site is too high or if there is no matching local SFI, SGHP starts forwarding to other sites using SALVE.



(a) Success rate over server capacity



(b) Service quality over server capacity



(c) Req. capacity over target success rate

Fig. 3: Single site scenario, running centralized scheduling IA-MPP vs baselines. Varying server capacity $c_S$ to reach full success rate. $c_S$ normalized to IA-MPP's $c_S$ at 100% success rate.

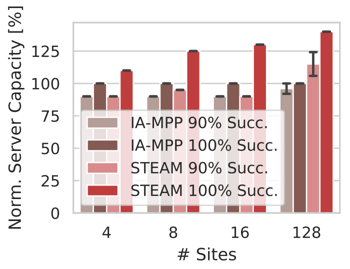
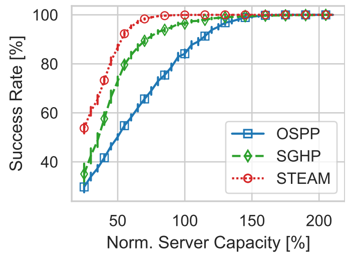
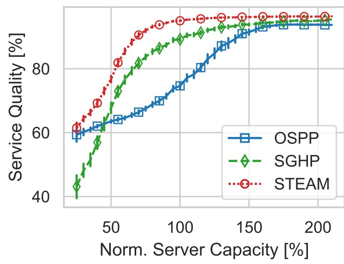


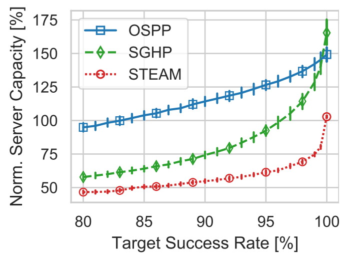
Fig. 4: From centralized to distributed scheduling, varying #sites. Normalized to IA-MPP.



(a) Success rate over server capacity



(b) Service quality over server capacity



(c) Req. capacity over target success rate

Fig. 5: 50 sites running distributed scheduling: STEAM vs baselines. Varying server capacity $c_S$ to reach full success rate. $c_S$ normalized to STEAM's $c_S$ at 100% success rate.

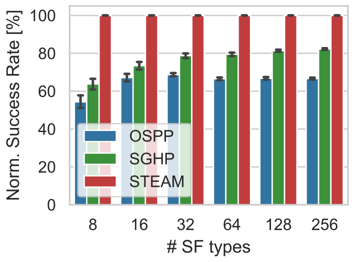


Fig. 6: 50 sites, varying #SFs, using $c_S$ within 50%-100% of STEAM's $c_S$ with full success.

## B. Setup

Unless stated otherwise, STEAM uses $\theta_l = 0.1$, $\theta_h = 1.3$, $\phi_b = 1$. We measure the performance of the schedulers when running the servers at a certain capacity $c_s$. Sweeping $c_s$ allows to draw conclusions of how effectively the schedulers are able to leverage all available processing power. All scenarios use link latencies following a Poisson distribution with $700\mu s$ for SFI-SFF links and $3000\mu s$ for SFF-SFF links [40], [41]. We repeat each experiment with five different seeds.

*1) Metrics.* We study two performance metrics: **Success rate** is the ratio of successfully served packets to the total number of arrivals. **Service quality** is one minus the total latency (ingress-egress) of a packet normalized to the QoS deadline of its SFC, also called "average response latency" [7]. The higher the values for these metrics, the better the solution.

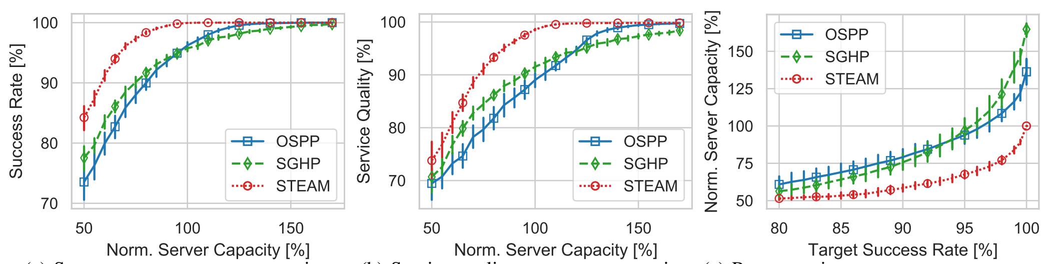
*2) Workload.* Unless stated otherwise, the experiments use a configuration as follows. The flow arrivals are time-varying and bursty. We use a Markov modulated process (MMP) [42] to simulate flow arrivals, which is a widely used model [43]–[45], with two states – "low" and "high". $\lambda_l$ and $\lambda_h$ are the flow arrival rates in these respective states, $p_l$ is the probability of transition from low to high state, and $p_h$ the opposite. We use $p_l = 0.56$, $p_h = 0.4$, $\lambda_h = 1/240\mu s$, $\lambda_l = 1/24\mu s$. We consider the packet arrival process within a flow to be random and independent from other flows, following a Poisson distribution ($\lambda_f = 1/800\mu s$). Flow sizes are also random, following a Poisson distribution ($\lambda_s = 150\mu s$). Each flow randomly selects an existing SFC and a pair of ingress/egress SFFs. Each SFC has a QoS deadline, set as a function of the service rates of involved SFs, which specifies the maximum allowed latency observed by a packet (typically $\ll 100ms$). We consider the SFI processing rates to be similar to the numbers reported for NFVs [26], [39], [46], in particular to values in the range of $1s/82\mu s$ - $1s/200\mu s$ per resource unit (cf. § II).

## C. Single-Site Experiments

We first consider a single-site topology with 1 SFF, 36 servers, 5 SFs and 80 SFIs. There are five SFCs: $C_1 = (F_1, F_2)$, $C_2 = (F_1, F_3, F_5)$, $C_3 = (F_2, F_4)$, $C_4 = (F_5)$, and $C_5 = (F_3, F_4)$ with QoS deadlines $\{56, 100, 44.4, 28, 56.4\}ms$. Fig. 3 shows the results for IA-MPP and the two competing heuristics running a single site, so all schedulers have access to all state, making comparison fair. We normalized server capacities to the capacity required by IA-MPP to achieve full success. We observe that IA-MPP outperforms the baselines, even in a non-distributed scenario. Specifically, we observe from Fig. 3a and Fig. 3b that IA-MPP provides the best success rate and quality of service, given a server capacity, while OSPP shows the worst performance. Fig. 3c depicts the required capacity to achieve success rates above 80%. To achieve 0 packet drops, an OSPP solution needs twice the capacity(!), and SGHP 25% more server capacity. These results illustrate that using IA-MPP reduces required server capacity to achieve a target success rate, while also providing better packet latency.

## D. Multi-Site Experiments

*1) **IA-MPP vs STEAM**.* First we study the effect of distributing the scheduling decisions per SFF. We sweep the number of sites from 4 to 128 (and traffic load accordingly),



(a) Success rate over server capacity (b) Service quality over server capacity (c) Req. capacity over target success rate

Fig. 7: Pcap workload: STEAM vs baselines. Varying server capacity $c_S$ to reach full success rate. $c_S$ normalized to STEAM's $c_S$ at $100\%$ success rate.

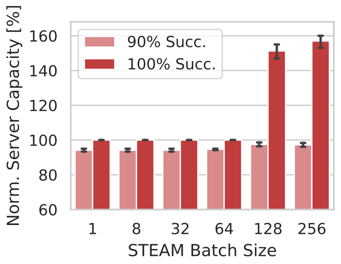
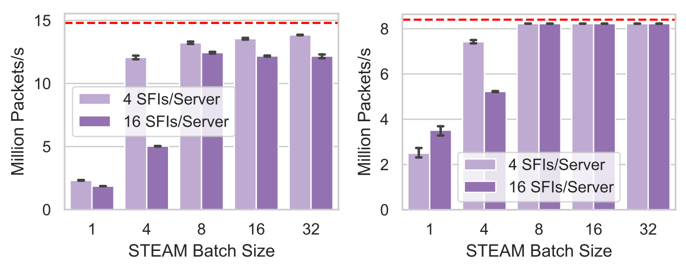


Fig. 8: Performance effect on success rate of varying batch size running STEAM.



(a) Goodput at 64B packets workload (b) Goodput at 128B packets workload

Fig. 9: STEAM prototype scheduling performance; varying $\phi_b$

and use for each site the same configuration as in § V-C. Fig. 4 compares the required server capacity to reach $90\%$ and $100\%$ success rate running STEAM vs IA-MPP. The values are normalized within each site to the capacity required by IA-MPP to achieve full success. Note that each STEAM instance uses only site-local state. We observe that the performance gap between STEAM and IA-MPP increases as we increase the size of the network or the required success rate target. For smallest topology, the two perform almost identically, but the gap increases to $40\%$ when using a 32 times larger topology, hence this gap grows slower compared to the topology size increase. Nevertheless, STEAM shows great performance, considering the fact that IA-MPP runs global optimization.

*2) **Performance at scale**.* Next we consider a topology in the image of publicly available information on data center locations of an Internet service provider (ISP) [41]. The topology comprises 50 sites, each with one SFF and 6 to 12 servers. There are 10 SFs in the network with a total of 1600 SFIs across all sites and 30 SFCs each with up to four SFs. We compare STEAM with baseline solutions, all using only site-local state. Fig. 5 depicts the results. STEAM shows best performance, reaching full success with $50-70\%$ less server capacity. This is as STEAM, driven by our optimal solution, tries to maximize the resource multiplexing in the network and hence can efficiently use available resources. Furthermore, better service quality signals better packet latency with STEAM.

*3) **Complexity increases**.* Next we vary the number of SFs in order to make the scheduling problem more challenging. The more SFs in the network, the more complex the problem becomes for STEAM, hence scheduling decisions might be negatively affected. We use again the ISP setup with 50 sites and set the number of total SFIs to 20 times the number of SFs, and create 3 times as many SFCs as SFs available. Fig. 6 shows the average success rates when running servers at capacities between $50\%$ to $100\%$ of the capacity level which STEAM needs to achieve full success. We see that STEAM's gain in success rate over baselines remains always above $20\%$ - $35\%$.

*4) **Trace-driven workload**.* In this experiment we use real-world trace files of a related scenario capturing, end-to-end voice and video Skype calls with a total of 484 nodes [47]. We consider a topology of 10 sites, 5 SFs, 5 SFCs, 100 SFIs in total, and 4 servers per site, so that each SFF receives the traffic from $\sim$ 48 Skype nodes. For each seed we take a $10min$ slice from the trace, which we consider to be a reasonable period between two offline planning phases. We consider packets with same source and destination addresses to belong to the same flow and apply the same SFC. Fig. 7 shows similar results as when running the MMP-based workload. STEAM shows best success rates and service quality at all shown server capacities. Using STEAM reduces the amount of server resources needed for full success by $30\%$ - $70\%$ compared to the baselines. These results indicate that gains are not due to specific tuning of the traffic model, but hold across different traffic patterns.

*5) **Batch scheduling**.* Finally, we study the effect of batch size $\phi_b$ on STEAM's performance. Batch scheduling lowers time complexity, but might negatively affect overall scheduling decisions. We study the trade-off. Fig. 8 shows the required capacity to achieve $90\%$ and $100\%$ success rate when varying $\phi_b$ of STEAM. Values are normalized to the server capacity required when running STEAM with $\phi_b = 1$ (no batching) and reaching full success. Up to $\phi_b = 64$, there is no significant drawback to batching. With $\phi_b$ of 128 or 256, $90\%$ target success rate requires slightly more server capacity; to reach full success, we require $50\%$ - $60\%$ more server capacity. Next we show how batching makes runtime scheduling feasible.

## *E. Prototype*

As described in § IV-B, we have implemented a prototype of STEAM based on DPDK [21], including the NSH protocol [33] to check feasibility running on a standard server with varying bucket sizes $\phi_b$. We use two servers (each $2\times$ E5-2630,

$128GB$ memory, Intel X520-2 10G SFP+; Linux 4.15.0-48-generic; DPDK 18.11.1) connected via a switch. One of the servers runs our packet generator (a FastClick [48] module), and the other runs STEAM. STEAM uses one core for receiving packets and running SALVE (we set $\theta_l = \theta_h = \infty$, to force all packets going to STEAM), and one core for running STEAM's scheduler sending packets back to the packet generator (intentionally to the SFIs). For each SFF buffer we use a DPDK ring buffer of up to 2048 packets. The system uses 16 hugepages of $1GB$ each, shared among the ring buffers. Note that we did not configure special optimizations, e.g., distributing the buffers across multiple Rx cores.

We run experiments with packets of size $64B$ and $128B$, which corresponds to $\sim 14.88$ and $\sim 8.45$ $10^6/s$ packets, respectively. The packet generator sends packets at line rate to STEAM and receives packets from STEAM after each scheduling decision. We report the rate at which the traffic generator receives packets from STEAM. To test the effect of scheduling complexity, we run the experiments with 4 and 16 SFIs per server. Fig. 9 shows the packet rate STEAM can uphold, v.s. the (theoretical) hardware limit of the NIC (red). STEAM reaches almost line rate starting from a batch size of 8, which translates to 2 up to 3.8 $10^6/s$ scheduling decisions combined with batching, STEAM hits almost $15{*}10^6/s$ packets. For packets of size $64B$, we do not hit the NIC limit, as we did not apply all possible micro-optimizations.

## VI. Related Work

In short, our work differs from all previous works related to virtualized network function (VNF) placement/scheduling in one or more of following aspects: (1) We consider *runtime* traffic scheduling *without a priori knowledge* of traffic distribution. (2) We target global optimization as a distributed scheduling problem, assuming *no complete view of the system*. (3) Scheduling decisions are made at *packet-level vs flow-level*, making our solution more adaptive to traffic dynamics.

Some recent works [31], [49]–[51] consider optimizations and scheduling at the level of a *single server or CPU core*. In particular, NFVnice is a VNF framework for CPUs that aims for fair and efficient resource allocation of chains, considering the impact of different VNFs on resource usage. Katsikas et al. [31] propose an intertwined setup of network devices and servers, allowing to reduce inter-core transfers of packets on the server and by this improving single-server VNFs throughput. Meng et al. [50] split an SFC into smaller semantically equivalent VNFs, enabling reuse of parts of an SFC across others.

Many research efforts have recently targeted network-wide VNF scheduling, inside a single data center or across multiple data centers. Nevertheless, most of them consider centralized SFC/VNF scheduling, assuming a scheduler with *global knowledge of the network* and often statistical information on traffic distribution. Mechtri et al. [52] consider joint placement and scheduling of SFCs for infrastructures mapping to undirected graphs, using *a priori* knowledge of the required static bandwidth of each network flow. Similar problems are investigated by others [10]–[15], [18]. Assuming perfect knowledge about traffic volumes, these placement/scheduling solutions can be applied only offline, or take decisions ahead of flow arrival. Qu et al. [16] consider dynamic flow demands, but allow a server to run only one SF instance and a link to forward only traffic from one flow at a time. Eramo et al. [17] allow traffic to change while being processed, but still require knowledge of flows' nominal and maximum traffic volumes. Anwer et al. [18] use the input and output traffic volume of all SFIs in the system to dynamically update the routing of SFC.

Different optimization objectives have been considered for VNF placement and scheduling [8], [39], [53]–[55]. Marotta et al. [53] tackle energy-efficiency in SFCs placement and scheduling, minimizing the number of involved switches and servers. Caggiani et al. [54] focus on internal switching of VNFs on a server to reduce the total switching overhead of an SFC. However, networking cost (e.g., latency) is neglected. Fei et al. [39] consider demand prediction for VNFs, based on which VNFs are scaled up by adding new instances and traffic is split among instances, with the objective of minimizing prediction error and system configuration cost. Yikai et al. [55] uses deep learning to reduce the cost of involved servers, but employs coarse grained scheduling per SF.

Some earlier work related to fair queuing is also relevant here. Bennet and Zhang [56] propose hierarchical fair queuing to provide network load balancing by scheduling packet flows over available paths. The proposed solution requires *a priori* knowledge of each flow type's share of assigned resources and arrival rates. Stoica et al. [57] use predictions of arrival rates of flow types to decide on the share of resources which should be assigned to each flow type and the corresponding link that a packet should be scheduled over. These solutions therefore fall into the same category as the other proposals mentioned above. Besides, they do not consider any chaining of SFs.

The works most closely related to ours are those of (1) Bhamara et al. [58] and (2) Satyam et al. [8]. (1) applies queuing models for servers and links in a multi-cloud environment to minimize inter-cloud traffic and response time. (2) studies VNF placement and CPU allocation for co-located VNFs in 5G networks to minimize end-to-end traffic latency. Both (1) and (2) assume *a priori* knowledge of packet arrival rates.

## VII. Conclusions

We proposed a runtime SFC scheduling policy, which can be deployed in a distributed manner, and demonstrated that, given fixed resource capacities, it can achieve significantly higher success rates and better service quality than existing static or coarse-grained solutions. It thus decreases the amount of resources in the network that need to be allocated to provide a target quality of service guarantee. Further avenues include (1) We assumed that SFIs are stateless. Although a realistic assumption in many scenarios [26], [27], it might not hold in others. We thus aim to study stateful SFIs. (2) We are keen to study the effect of admission control on the performance. (3) We aim to apply machine learning solutions to the problem.

## REFERENCES

[1] National Science Foundation, "US-EU Internet Core & Edge Technologies (ICE-T)," *NSF 18-535*, https://www.nsf.gov/pubs/2018/nsf18535/nsf18535.pdf, May 2018.

[2] European Commission, "ICT-20-2019-2020: 5G Long Term Evolution," *Horizon 2020 - Information and Communication Technologies, European Commission Decision C(2019)4575*, https://ec.europa.eu/research/participants/data/ref/h2020/wp/2018-2020/main/h2020-wp1820-leit-ict_en.pdf, July 2019.

[3] IETF, "Service Function Chaining Use Cases in Mobile Networks," Tech. Rep. draft-ietf-sfc-use-case-mobility-09, Jan. 2019.

[4] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz, "Service function chaining in next generation networks: State of the art and research challenges," *Communications Magazine*, vol. 55, no. 2, pp. 216–223, 2017.

[5] A. Alim *et al.*, "FLICK: Developing and Running Application-Specific Network Services," *ATC*, 2016.

[6] R. Stoenescu *et al.*, "In-Net: in-network processing for the masses," *EuroSys*, 2015.

[7] Q. Zhang, F. Liu, and C. Zeng, "Adaptive interference-aware VNF placement for service-customized 5g network slices," *INFOCOM*, 2019.

[8] A. Satyam, M. Francesco, C. F. Chiasserini, and D. Swedes, "Joint VNF Placement and CPU Allocation in 5G," *INFOCOM*, 2018.

[9] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, 2015.

[10] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," *INFOCOM*, 2015.

[11] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," *ACM CloudNet*, 2015.

[12] B. Martini, F. Paganelli, P. Cappanera, S. Turchi, and P. Castoldi, "Latency-aware composition of virtual functions in 5G," *NetSoft*, 2015.

[13] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," *NetSoft*, 2015.

[14] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta, "Joint optimization of service function chaining and resource allocation in network function virtualization," *IEEE Access*, vol. 4, pp. 8084–8094, 2016.

[15] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," *INFOCOM*, 2016.

[16] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *TCOM*, vol. 64, no. 9, pp. 3746–3758, 2016.

[17] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, "An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures," *TON*, vol. 25, no. 4, pp. 2008–2025, 2017.

[18] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming Slick Network Functions," *SIGCOMM*, 2015.

[19] S. Palkar *et al.*, "E2: a framework for nfv applications," *SOSP*, 2015.

[20] R. J. Williams, "Stochastic Processing Networks," *Annual Review of Statistics and Its Application*, vol. 3, no. 1, pp. 323–345, 2016.

[21] Intel, "Data plane development kit," https://www.dpdk.org.

[22] Cisco, "Best Practices in Core Network Capacity Planning," Tech. Rep.

[23] R. Nadiv and T. Naveh, "Wireless backhaul topologies: Analyzing backhaul topology strategies," *White Paper Ceragon*, 2010.

[24] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," *SIGCOMM*, 2017.

[25] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," *SIGCOMM*, 2017.

[26] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," *NSDI*, 2017.

[27] 3GPP, "Technical Realization of Service Based Architecture; Stage 3," 3GPP, Technical Specification (TS) TS29.500, 2018, version 0.4.0.

[28] J. M. Harrison, "Brownian models of open processing networks: canonical representation of workload," *The Annals of Applied Probability*, vol. 10, no. 1, pp. 75–103, 2000.

[29] J. G. Dai and W. Lin, "Maximum Pressure Policies in Stochastic Processing Networks," *Operations Research*, vol. 53, no. 2, pp. 197–218, 2005.

[30] ——, "Asymptotic optimality of maximum pressure policies in stochastic processing networks," *The Annals of Applied Probability*, vol. 18, no. 6, pp. 2239–2299, 2008.

[31] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," *NSDI*, 2018.

[32] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[33] P. Quinn, U. Elzur, and C. Pignataro, "Network Service Header (NSH)," RFC 8300, 2018.

[34] M. Bramson and R. J. Williams, "Two Workload Properties for Brownian Networks," *Queueing Systems*, vol. 45, no. 3, pp. 191–221, 2003.

[35] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," *CCR*, vol. 44, no. 4, 2014.

[36] A. Saeed, Y. Zhao, N. Dukkipati, E. W. Zegura, M. H. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and Flexible Software Packet Scheduling," *NSDI*, 2019.

[37] T. Nelson, N. DeMarinis, T. A. Hoff, R. Fonseca, and S. Krishnamurthi, "Switches are monitors too!: Stateful property monitoring as a switch design criterion," *HotNets*, 2016.

[38] D. Firestone *et al.*, "Azure accelerated networking: Smartnics in the public cloud," *NSDI*, 2018.

[39] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF Scaling and Flow Routing with Proactive Demand Prediction," *INFOCOM*, 2018.

[40] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," *CCR*, vol. 45, no. 4, 2015.

[41] Cogent Communications, "Cogent Network Map," http://cogentco.com/en/network/network-map.

[42] W. Fischer and K. S. Meier-Hellstern, "The Markov-Modulated Poisson Process Cookbook," *Perform. Eval.*, vol. 18, no. 2, pp. 149–171, 1993.

[43] M. J. Neely, "Delay Analysis for Maximal Scheduling With Flow Control in Wireless Networks With Bursty Traffic," *TON*, vol. 17, no. 4, pp. 1146–1159, 2009.

[44] K. Wang, M. Lin, F. Ciucu, A. Wierman, and C. Lin, "Characterizing the Impact of the Workload on the Value of Dynamic Resizing in Data Centers," *Perform. Eval.*, vol. 85, pp. 1–18, 2015.

[45] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson, "Markovian Workload Characterization for QoS Prediction in the Cloud," *IEEE CLOUD*, 2011.

[46] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," *ACM CloudNet*, 2015.

[47] Telecommunications Networks Group - Politecnico di Torino, "Traces from Real Internet Traffic," http://tstat.polito.it/traces-skype.shtml.

[48] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," *IEEE/ACM ANCS*, 2015.

[49] S. G. Kulkarni *et al.*, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," *SIGCOMM*, 2017.

[50] Z. Meng, J. Bi, C. Sun, H. Wang, and H. Hu, "CoCo: Compact and Optimized Consolidation of Modularized Service Function Chains in NFV," *ICC*, 2018.

[51] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "SNF: Synthesizing high performance NFV service chains," *PeerJ Computer Science*, vol. 2, p. e98, 2016.

[52] M. Mechtri, C. Ghribi, and D. Zeghlache, "A scalable algorithm for the placement of service function chains," *TNSM*, vol. 13, no. 3, pp. 533–546, 2016.

[53] A. Marotta and A. Kassler, "A power efficient and robust virtual network functions placement problem," *ITC*, vol. 1, 2016.

[54] M. C. Luizelli, D. Raz, and Y. Sa'ar, "Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching," *INFOCOM*, 2018.

[55] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "NFVdeep: adaptive online service function chain deployment with deep reinforcement learning," *IEEE/ACM IWQoS*, 2019.

[56] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *TON*, vol. 5, no. 5, pp. 675–689, 1997.

[57] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," *CCR*, vol. 28, no. 4, pp. 118–130, 1998.

[58] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Computer Communications*, vol. 102, pp. 1–16, 2017.