# Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives

Tamás Lévai, *Budapest University of Technology and Economics & University of Southern California;* Felicián Németh, *Budapest University of Technology and Economics;* Barath Raghavan, *University of Southern California;* Gábor Rétvári, *MTA-BME Information Systems Research Group & Ericsson Research, Hungary*

https://www.usenix.org/conference/nsdi20/presentation/levai

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

# Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives

Tamás Lévai[1,2], Felicián Németh[1], Barath Raghavan[2], and Gábor Rétvári[3,4]
[1]*Budapest University of Technology and Economics*
[2]*University of Southern California*
[3]*MTA-BME Information Systems Research Group*
[4]*Ericsson Research, Hungary*

## Abstract

Data flow graphs are a popular program representation in machine learning, big data analytics, signal processing, and, increasingly, networking, where graph nodes correspond to processing primitives and graph edges describe control flow. To improve CPU cache locality and exploit data-level parallelism, nodes usually process data in batches. Unfortunately, as batches are split across dozens or hundreds of parallel processing pathways through the graph they tend to fragment into many small chunks, leading to a loss of batch efficiency.

We present Batchy, a scheduler for run-to-completion packet processing engines, which uses controlled queuing to efficiently reconstruct fragmented batches in accordance with strict service-level objectives (SLOs). Batchy comprises a runtime profiler to quantify batch-processing gain on different processing functions, an analytical model to fine-tune queue backlogs, a new queuing abstraction to realize this model in run-to-completion execution, and a one-step receding horizon controller that adjusts backlogs across the pipeline. We present extensive experiments on five networking use cases taken from an official 5G benchmark suite to show that Batchy provides 2–3× the performance of prior work while accurately satisfying delay SLOs.

## 1 Introduction

One near-universal technique to improve the performance of software packet processing engines is batching: collect multiple packets into a single burst and perform the same operation on all the packets in one shot. Processing packets in batches is much more efficient than processing a single packet at a time, thanks to amortizing one-time operational overhead, optimizing CPU cache usage, and enabling loop unrolling and SIMD optimizations [7,8,11,22,26]. Batch-processing alone often yields a 2–5× performance boost. Fig. 1 presents a series of micro-benchmarks we performed in BESS [14] and FastClick [3], two popular software switches, showing that executing an ACL or a NAT function is up to 4 times

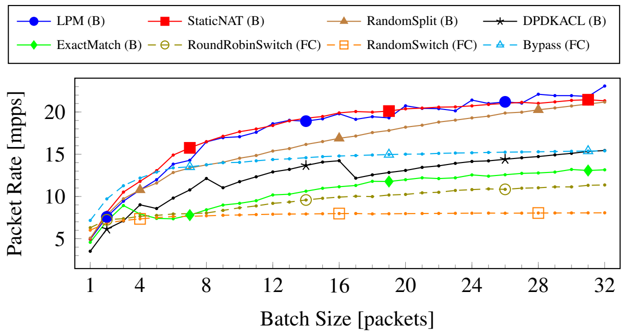


Figure 1: Maximum packet rate (in millions of packets per second, mpps) over different packet processing micro-benchmarks in BESS [14] (marked with B) and in FastClick [3] (FC) when varying the input batch size.

as efficient on batches containing 32 packets compared to single-packet batches. Prior studies provided similar batch-processing profiles in VPP [2,22], [27, Fig. 3]. Accordingly, batching is used in essentially all software switches (VPP [2, 27], BESS [14], FastClick [3], NetBricks [36], PacketShader [15], and ESwitch [30]), high-performance OS network stacks and dataplanes [3,6,8,11], user-space I/O libraries [1,16], and Network Function Virtualization platforms [19,21,42,45,50].

Unfortunately, even if the packet processing engine receives packets in bursts [1,2,6,27,29,41,50], batches tend to break up as they progress through the pipeline [20]. Such *batch-fragmentation* may happen in a multi-protocol network stack, where packet batches are broken into smaller per-protocol batches to be processed by the appropriate protocol modules (e.g., pure Ethernet, IPv4, IPv6, unicast/multicast, MPLS, etc.) [8,11]; by a load-balancer that splits a large batch into smaller batches to be sent to different backends/CPUs for processing [19]; in an OpenFlow/P4 match-action pipeline where different flow table entries may appoint different next-stage flow tables for different packets in a batch [30,38], or in essentially any packet processing engine along the branches of the data flow graph (splitters) [3,14,16,31,47]. In fact, any operation that involves *matching* input packets against a lookup

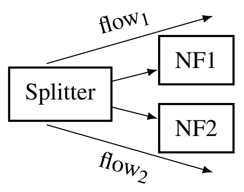


Figure 2: A sample packet processing data flow graph: a two-way splitter with two network functions (NFs). Both NFs incur one unit of execution cost per each processed batch and another one unit per each packet in the batch; the splitter incurs negligible cost. There are two service chains (or flows), one taking the upper and one taking the lower path.

table and *distributing* them to multiple next-stage processing modules may cause packet batches to fragment [20]. Worse still, fragmentation at subsequent match-tables combine multiplicatively; e.g., VRF-splitting on 16 VLANs followed by an IP lookup on 16 next-hops may easily break up a single batch of 256 packets into 256 distinct batches containing one packet each. Processing packets in the resultant small chunks takes a huge toll on the compute efficiency of the pipeline, which was designed and optimized for batch-processing in the first place [2, 11, 27]. We stress that batch-fragmentation is quite dynamic, depending on the input traffic profile, the offered load, the packet processing graph, the NFs, and flow table configuration; therefore, modeling and quantifying the resultant performance loss is challenging [22].

Trivially, *fragmented batches can be "de-fragmented" using queuing*, which delays the execution of an operation until enough packets line up at the input [1, 6, 15, 50]. This way, processing occurs over larger batches, leading to potentially significant batch-processing gains. However, queuing packets, thereby artificially slowing down a pipeline in order to speed it up, is tricky [8, 11]; a suboptimal queuing decision can easily cause delay to skyrocket. Fig. 2 shows a motivating example: if two batches containing 2 packets each enter the pipeline then unbuffered execution incurs 8 units of execution cost, as the splitter breaks up each batch into two batches containing one packet each. Placing a queue at the NF inputs, however, enables recovery of the full batches, bringing the execution cost down to 6 units (1 unit per the single batch processed and 2 units per the 2 packets in the batch) but increasing delay to 2 full turnaround times. For a $k$-way splitter the cost of an unbuffered execution over $k$ batches including $k$ packets each would be $2k^2$, which buffering would reduce to $k+k^2$; about $2\times$ batch-processing gain at the cost of $k\times$ queuing delay.

Of course, packet processing cannot be delayed for an arbitrarily long time to recover batches in full, since this may violate the service level objectives (SLOs) posed by different applications. A typical tactile internet and robot control use case requires 1–10 msec one-way, end-to-end, 99th percentile latency [28]; the 5G radio access/mobile core requires 5–10 msec; and reliable speech/video transport requires delay below 100-200 msec. At the extreme, certain industry automation, 5G inter-base-station and antenna synchronization, algorithmic stock trading, and distributed memory cache applications limit the one-way latency to 10-100 $\mu$sec [12, 25].

The key observation in this paper is that *optimal batch-scheduling in a packet processing pipeline is a fine balancing act to control queue backlogs, so that processing occurs in as large batches as possible while each flow traversing the pipeline is scheduled just fast enough to comply with the SLOs*. We present Batchy, a run-to-completion batch-scheduler framework for controlling execution in a packet-processing pipeline based on strict service-level objectives. Our contributions in Batchy are as follows:

**Quantifying batch-processing gain.** We observe that batch-processing efficiency varies widely across different packet-processing functions. We introduce the *Batchy profiler*, a framework for quantifying the batched service time profile for different packet-processing functions.

**Analytical model.** We introduce an expressive mathematical model for SLO-based batch-scheduling, so that we can reason about the performance and delay analytically and fine tune batch de-fragmentation subject to delay-SLOs. We also fix the set of basic assumptions under which the optimal schedule is well-defined (see earlier discussion in [6, 15, 50]).

**Batch-processing in run-to-completion mode.** Taking inspiration from Nagle's algorithm [32], we introduce a new queuing abstraction, the fractional buffer, which allows us to control queue backlogs at a fine granularity even in run-to-completion scheduling, which otherwise offers very little control over when particular network functions are executed.

**Design, implementation, and evaluation of Batchy.** We present a practical implementation of our batch-scheduling framework and, taking use cases from an official 5G NFV benchmark suite (L2/L3 gateway with and without ACL, NAT, VRF, a mobile gateway, and a robot-control pipeline), we demonstrate that Batchy efficiently exploits batch-processing gain consistently across a wide operational regime with small controller overhead, bringing 1.5–3$\times$ performance improvement compared to outliers and earlier work [21], while satisfying SLOs. Batchy is available for download at [4].

The rest of the paper is structured as follows. In Section 2 we introduce the Batchy profiler, Section 3 presents the idealized mathematical model and introduces fractional buffers, Section 4 discusses our implementation in detail, and Section 5 describes our experiments on real-life use cases. Finally, Section 6 discusses related work and Section 7 concludes the paper. A detailed exposition of the algorithms used in our implementation is given in the Appendix.

## 2 Profiling Batch-processing Gain

There are many factors contributing to the efficiency of batch-processing; next, we highlight some of the most important

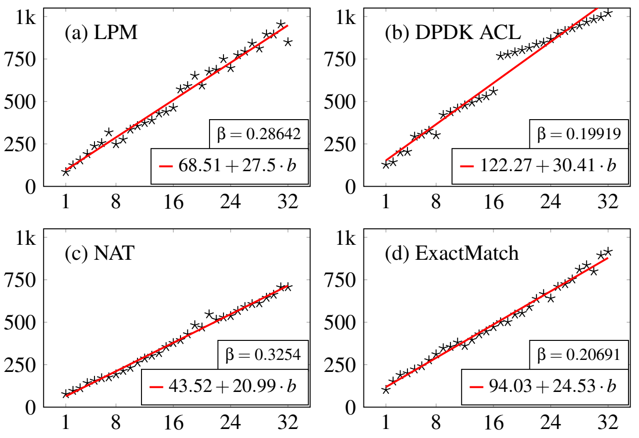


Figure 3: Service-time profile: execution time [nsec] for different modules as the function of the input batch size, averaged over 10 runs at 100,000 batches per second. The inset gives the batchiness $\beta_v$ and the linear regression $T_{v,0} + T_{v,1}b_v$. Observe the effects of quad-loop/SIMD optimization for the ACL module at batch size 4, 8, and 16.

ones [26,47]. First, executing a network function on a batch incurs non-negligible computational costs independent from the number of packets in it, in terms of CPU-interrupt, scheduling, function call, I/O, memory management, and locking overhead, and *batching amortizes this fixed-cost component* over multiple packets [8, 26]. Second, executing an operation on multiple packets in one turn *improves CPU cache usage*: packets can be prefetched from main memory ahead of time and data/code locality improves as CPU caches are populated by the first packet and then further processing happens without cache misses. For example, VPP modules are written so that the entire code fits into the instruction cache, reducing icache misses [2, 27]. Third, *loop unrolling*, a compiler optimization to rewrite loops into dual- or quad-loops [26] to improve branch predictor performance and keep the CPU pipeline full, is effective only if there are multiple packets to process in one shot. Batch-processing also opens the door to exploit data-level parallelism, whereby the CPU performs the same operation on a batch of 4–32 packets in parallel for the cost of a single SIMD instruction (SSE/AVX) [16].

Intuitively, different packet-processing functions may benefit differently from batch-processing; e.g., a module processing packets in a tight loop may benefit less than a heavily SIMD-optimized one. This will then affect batch-scheduling: reconstructing batches at the input of a lightweight module might not be worth the additional queuing delay, as there is very little efficiency gain to be obtained this way.

Fig. 3 provides a *service time profile* as the execution time for some select BESS modules [14] as the function of the batch size [22]. We observe two distinct execution time components. The *per-batch cost component*, denoted by $T_{v,0}$ [sec] for a module $v$, characterizes the constant cost that is incurred just for calling the module on a batch, independently from the number of packets in it. The *per-packet cost component* $T_{v,1}$, [sec/pkt], on the other hand, models the execution cost of each individual packet in the batch. A linear model seems a good fit for the service time profiles: accordingly we shall use the linear regression $T_v = T_{v,0} + T_{v,1}b_v$ [sec] to describe the execution cost of a module $v$ where $b_v$ is the batch-size, i.e., the average number of packets in the batches received by module $v$. The coefficient of determination $R^2$ is above 96% in our tests, indicating a good fit for the linear model.

The per-batch and per-packet components determine the potential batch-processing gain on different packet processing modules. We quantify this gain with the *batchiness* measure $\beta_v$, the ratio of the effort needed to process $B$ packets in a single batch of size $B$ through $v$ compared to the case when processing occurs in $B$ distinct single-packet batches:

$$\beta_v = \frac{T_{v,0} + B * T_{v,1}}{B(T_{v,0} + T_{v,1})} \sim \frac{T_{v,1}}{T_{v,0} + T_{v,1}} \text{ for large } B \ . \qquad (1)$$

Batchiness varies between 0 and 1; small $\beta_v$ indicates substantial processing gain on $v$ and hence identifies a potential control target. The relatively small batchiness measures in Fig. 3 suggest that most real-world packet-processing functions are particularly sensitive to batch size.

Batchy contains a built-in profiler that runs a standard benchmark on the system under test at the time of initialization, collects per-batch and per-packet service-time components for common NFs, and stores the results for later use.

## 3 Batch-scheduling in Data Flow Graphs

Next, we present a model to describe batch-based packet processing systems. The model rests on a set of simplifying assumptions, which prove crucial to reason about such systems using formal arguments. We adopt the terminology and definitions from BESS, but we note that the model is general enough to apply to most popular data flow graph packet-processing engines, like VPP [47], Click/FastClick [3, 31], NetBricks [36], or plain DPDK [16]; match-action pipelines like Open vSwitch [38] or ESwitch [30]; or to data flow processing frameworks beyond the networking context like TensorFlow [10] or GStreamer [43].

### 3.1 System model

**Data flow graph.** We model the pipeline as a directed graph $\mathcal{G} = (V, E)$, with modules $v \in V$ and directed links $(u, v) \in E$ representing the connections between modules. A *module* $v$ is a combination of a (FIFO) *ingress queue* and a *network function* at the egress connected back-to-back (see Fig. 4). *Input gates* (or ingates) are represented as in-arcs $(u, v) \in E$ : $u \in V$ and *output gates* (or outgates) as out-arcs $(v, u) \in E$ :

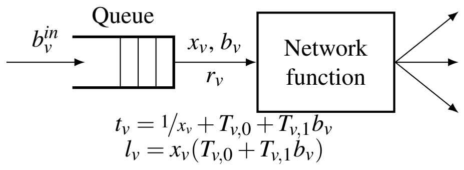

Queue
$b_v^{in}$
$x_v$, $b_v$
$r_v$
Network function
$t_v = 1/x_v + T_{v,0} + T_{v,1}b_v$
$l_v = x_v(T_{v,0} + T_{v,1}b_v)$

Figure 4: A Batchy module.

$u \in V$. A batch sent to an outgate $(v,u)$ of $v$ will appear at the corresponding ingate of $u$ at the next execution of $u$. Modules never drop packets; we assume that whenever an ACL module or a rate-limiter would drop a packet it will rather send it to a dedicated "drop" gate, so that we can account for lost packets. A normal queue is a module with an empty network function.

**Batch processing.** Packets are injected into the ingress, transmitted from the egress, and processed from outgates to ingates along data flow graph arcs, in batches [2,14,16,27,30]. We denote the maximum batch size by $B$, a system-wide parameter. For the Linux kernel and DPDK $B = 32$ or $B = 64$ are usual settings, VPP sets the batch size to 256 packets by default [27], while GPU/NIC offload often works with $B = 1024$ or even larger to maximize I/O efficiency [40,50].

**Splitters/mergers.** Any module may have multiple ingates (merger) and/or multiple outgates (splitter), or may have no ingate or outgate at all. An IP Lookup module would distribute packets to several downstream branches, each performing group processing for a different next-hop (splitter); a NAT module may multiplex traffic from multiple ingates (merger); and an IP Checksum module would apply to a single datapath flow (single-ingate–single-outgate). Certain modules are represented without ingates, such as a NIC receive queue; we call these *ingress modules* $S$. Similarly, a module with no outgates (e.g., a transmit queue) is an *egress module*.

**Compute resources.** A *worker* is an abstraction for a CPU core, where each worker $w \in \mathcal{W}$ is modeled as a connected subgraph $\mathcal{G}_w = (V_w, E_w)$ of $\mathcal{G}$ with strictly one ingress module $S_w = \{s_w\}$ executing on the same CPU. We assume that when a data flow graph has multiple ingress modules then each ingress is assigned to a separate worker, with packets passing between workers over double-ended queues. A typical setup is to dedicate a worker to each receive queue of each NIC and then duplicate the entire data flow graph for each worker. Each worker may run a separate explicit scheduler to distribute CPU time across the modules in the worker graph, or it may rely on run-to-completion; see Appendix A for an overview.

**Flows.** A flow $f = (p_f, R_f, D_f), f \in \mathcal{F}$ is our abstraction for a service chain, where $p_f$ is a path through $\mathcal{G}$ from the flow's ingress module to the egress module, $R_f$ denotes the offered packet rate at the worker ingress, and $D_f$ is the delay-SLO, the maximum permitted latency for any packet of $f$ to reach the egress. What constitutes a flow, however, will be use-case specific: in an L3 router a flow is comprised of all traffic destined to a single next-hop or port; in a mobile gateway a flow is a complex combination of a user selector and a bearer selector; in a programmable software switch flows are completely configuration-dependent and dynamic. Correspondingly, flow-dissection in a low-level packet processing engine cannot rely on the RSS/RPS function supplied by the NIC, which is confined to VLANs and the IP 5-tuple [6, 19]. Rather, in our framework flow dispatching occurs *intrinsically* as part of the data flow graph; accordingly, we presume that match-tables (splitters) are set up correctly to ensure that the packets of each flow $f$ will traverse the data flow graph along the path $p_f$ associated with $f$.

## 3.2 System variables

We argue that at multiple gigabits per second it is overkill to model the pipeline at the granularity of individual packets [22]. Instead, in our model variables are continuous and differentiable, describing system *statistics* over a longer period of time that we call the *control period*. This is analogous to the use of standard (continuous) network flow theory to model packet routing and rate control problems. We use the following variables to describe the state of the data flow graph in a given control period (dimensions indicated in brackets).

**Batch rate** $x_v$ [$1/s$]: the number of batches per second entering the network function in module $v$ (see again Fig. 4).

**Batch size** $b_v$ [pkt]: the average number of packets per batch at the input of the network function in module $v$, where $b_v \in [1,B]$ and, recall, $B$ is the maximum allowed batch size.

**Packet rate** $r_v$ [$\text{pkt}/s$]: the number of packets per second traversing module $v$: $r_v = x_v b_v$.

**Maximum delay** $t_v$ [sec]: delay contribution of module $v$ to the total delay of packets traversing it. We model $t_v$ as

$$t_v = t_{v,\text{queue}} + t_{v,\text{svc}} = 1/x_v + (T_{v,0} + T_{v,1} b_v) \quad , \tag{2}$$

where $t_{v,\text{queue}} = 1/x_v$ is the queuing delay by Little's law and $t_{v,\text{svc}} = T_{v,0} + T_{v,1} b_v$ is the service time profile (see Section 2).

**System load** $l_v$ (dimensionless): the network function in module $v$ with service time $t_{v,\text{svc}}$ executed $x_v$ times per second incurs $l_v = x_v t_{v,\text{svc}} = x_v(T_{v,0} + T_{v,1} b_v)$ system load in the worker.

**Turnaround-time** $T_0$ [sec]: the maximum CPU time the system may spend pushing a single batch through the pipeline. The turnaround time typically varies with the type and number of packets in each batch, the queue backlogs, etc.; correspondingly, we usually consider the time to execute *all* modules on maximum sized batches as an upper bound:

$$T_0 \le \sum_{v \in V} (T_{v,0} + T_{v,1} B) \quad . \tag{3}$$

## 3.3 Assumptions

Our aim is to define the simplest possible batch-processing model that still allows us to reason about flows' packet rate and maximum delay, and modules' batch-efficiency. The below assumptions will help to keep the model at the minimum; these assumptions will be relaxed later in Section 4.

**Feasibility.** We assume that the pipeline runs on a single worker and this worker has enough capacity to meet the delay-SLOs. In the next section, we show a heuristic method to decompose a data flow graph to multiple workers in order to address SLO violations stemming from inadequate resources.

**Buffered modules.** We assume that all modules contain an ingress queue and all queues in the pipeline can hold up to at most $B$ packets at any point in time. In the next section, we show how to eliminate useless queues in order to remove the corresponding latency and processing overhead.

**Static flow rate.** All flows are considered constant-bit-rate (CBR) during the control period (usually in the millisecond time frame). This assumption will be critical for the polynomial tractability of the model. Later on, we relax this assumption by incorporating the model into a receding-horizon optimal control framework.

### 3.4 Optimal explicit batch-schedule

Workers typically run an *explicit scheduler* to distribute CPU time across the modules in the worker graph. Common examples include Weighted Fair Queueing (WFQ) and Completely Fair Scheduling (CFS); here, the user assigns integer weights to modules and the scheduler ensures that runtime resource allocation will be proportional to modules' weight [46]. For simplicity, we consider an *idealized* WFQ/CFS scheduler instead, where execution order is defined in terms per-module *rates* and not weights; rates will be converted to weights later.

The idealized scheduler runs each module precisely at the requested rate. When scheduled, the modules' network function dequeues at most a single batch worth of packets from the ingress queue, executes the requested operation on all packets of the batch, forms new sub-batches from processed packets and places these to the appropriate outgates.

In this setting, we seek for a set of rates $x_v$ at which each module $v \in V$ needs to be executed in order to satisfy the SLOs. If multiple such rate allocations exist, then we aim to choose the one that minimizes the overall system load.

Recall, executing $v$ exactly $x_v$ times per second presents $l_v = x_v t_{v,\text{svc}} = x_v(T_{v,0} + T_{v,1} b_v)$ load to the system. The objective function, correspondingly, is to find rates $x_v$ that minimize the total system load $\sum_{v \in V} l_v$, taken across all modules:

$$\min \sum_{v \in V} x_v (T_{v,0} + T_{v,1} b_v) \quad . \tag{4}$$

Once scheduled, module $v$ will process at most $b_v \in [1, B]$ packets through the network function, contributing $t_{v,\text{svc}} = T_{v,0} + T_{v,1} b_v$ delay to the total latency of each flow traversing it. In order to comply with the delay-SLOs, for each flow $f$ it must hold that the total time spent by any packet in the worker ingress queue, plus the time needed to send a packet through the flow's path $p_f$, must not exceed the delay requirement $D_f$ for $f$. Using that the ingress queue of size $B$ may develop a backlog for only at most one turnaround time $T_0$ (recall, we assume there is a single worker and each queue holds at most $B$ packets), and also using (2), we get the following *delay-SLO constraint*:

$$t_f = T_0 + \sum_{v \in p_f} \left( {}^{1}\!/_{x_v} + T_{v,0} + T_{v,1} b_v \right) \le D_f \qquad \forall f \in \mathcal{F} \quad . \tag{5}$$

Each module $v \in V$ must be scheduled frequently enough so that it can handle the *total offered packet rate* $R_v = \sum_{f: v \in p_f} R_f$, i,e., the sum of the requested rate $R_f$ of each flow $f$ traversing $v$ (recall, we assume flow rates $R_f$ are constant). This results the following *rate constraint*:

$$r_v = x_v b_v = \sum_{f: v \in p_f} R_f = R_v \qquad \forall v \in V \quad . \tag{6}$$

Finally, the backlog $b_v$ at any of the ingress queues across the pipeline can never exceed the queue size $B$ and, of course, all system variables must be non-negative:

$$1 \le b_v \le B, \; x_v \ge 0 \qquad \forall v \in V \quad . \tag{7}$$

Together, (4)–(7) defines an optimization problem which provides the required static scheduling rate $x_v$ and batch size $b_v$ for each module $v$ in order to satisfy the SLOs while maximizing the batch-processing gain. This of course needs the turnaround time $T_0$; one may use the approximation (3) to get a conservative estimate. Then, substituting $b_v = \sum_{f: v \in p_f} {}^{R_f}\!/_{x_v} = {}^{R_v}\!/_{x_v}$ using (6), we get the following system, now with only the batch-scheduling rates $x_v$ as variables:

$$\min \sum_{v \in V} x_v \left(T_{v,0} + T_{v,1} \frac{R_v}{x_v}\right) \tag{8}$$

$$t_f = T_0 + \sum_{v \in p_f} \left( \frac{1}{x_v} + T_{v,0} + T_{v,1} \frac{R_v}{x_v} \right) \le D_f \quad \forall f \in \mathcal{F} \tag{9}$$

$${}^{R_v}\!/_{B} \le x_v \le R_v \qquad \forall v \in V \tag{10}$$

Since the constraints and the objective function are convex, we conclude that (8)–(10) *is polynomially tractable and the optimal explicit batch-schedule is unique* [5]. Then, setting the scheduling weights proportionally to rates $x_v$ results in the optimal batch-schedule on a WFQ/CFS scheduler [46].

### 3.5 Run-to-completion execution

WFQ/CFS schedulers offer a plausible way to control batch de-fragmentation via the per-module weights. At the same time, often additional tweaking is required to avoid head-of-line blocking and late drops along flow paths [21], and even running the scheduler itself may incur non-trivial runtime overhead. *Run-to-completion execution*, on the other hand, eliminates the explicit scheduler all together, by tracing the entire input batch though the data flow graph in one shot without the risk of head-of-line blocking and internal packet drops [6,14]. Our second batch-scheduler will therefore adopt run-to-completion execution.

The idea in run-to-completion scheduling is elegantly simple. The worker checks the input queue in a tight loop and, whenever the queue is not empty, it reads a single batch and injects it into pipeline at the ingress module. On execution, each module will process a single batch, place the resulting packets at the outgates potentially breaking the input batch into multiple smaller output batches, and then recursively schedule the downstream modules in order to consume the sub-batches from the outgates. This way, the input batch proceeds through the entire pipeline in a single shot until the last packet of the batch completes execution, at which point the worker returns to draining the ingress queue. Since upstream modules will *automatically* schedule a downstream module whenever there is a packet waiting to be processed, run-to-completion execution does not permit us to control when individual modules are to be executed. This makes it difficult to enforce SLOs, especially rate-type SLOs, and to delay module execution to de-fragment batches.

Below, we introduce a new queuing abstraction, the *fractional buffer*, which nevertheless lets us exert fine-grained control over modules' input batch size. The fractional buffer is similar to Nagle's algorithm [32], originally conceived to improve the efficiency of TCP/IP networks by squashing multiple small messages into a single packet. The backlog is controlled so as to keep end-to-end delay reasonable. Indeed, Nagle's algorithm exploits the same batch-efficiency gain over the network as we intend to exploit in the context of compute-batching, motivating our choice to apply it whenever there is sufficient latency slack available.

A fractional buffer maintains an internal FIFO queue and exposes a single parameter to the control plane called the *trigger* $b$, which enables tight control of the queue backlog and thereby the delay. The buffer will enqueue packets and suppress execution of downstream modules until the backlog reaches $b$, at which point a packet batch of size $b$ is consumed from the queue, processed in a single burst through the succeeding module, and execution of downstream modules is resumed. Detailed pseudocode is given in Appendix C.

We intentionally define the trigger in the batch-size domain and not as a perhaps more intuitive timeout [32], since timeouts would re-introduce an explicit scheduler into the otherwise "schedulerless" design. Similarly, we could in theory let the buffer to emit a batch larger than $b$ whenever enough packets are available; we intentionally restrict the output batch to size $b$ so as to tightly control downstream batch size.

What remains is to rewrite the optimization model (8)–(10) from explicit module execution rates $x_v$ to fractional buffer triggers. Interestingly, *jumping from rate-based scheduling to the run-to-completion model is as easy as substituting variables*: if we replace the ingress queue with a fractional buffer with trigger $b_v$ in each module $v$, then the subsequent network function will experience a batch rate of $x_v = R_v/b_v$ at batch size $b_v$. Substituting this into the optimization problem (4)–(7) yields the *optimal batch-schedule for the run-to-completion model* with variables $b_v : v \in V$:

$$\min \sum_{v \in V} \frac{R_v}{b_v}(T_{v,0} + T_{v,1} b_v) \tag{11}$$

$$t_f = T_0 + \sum_{v \in p_f} \left( \frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v \right) \le D_f \quad \forall f \in \mathcal{F} \tag{12}$$

$$1 \le b_v \le B \qquad \forall v \in V \tag{13}$$

Again, the optimization problem is convex and the optimal setting for the fractional buffer triggers is unique. In addition, the feasible region is linear, which makes the problem tractable for even the simplest of convex solvers. Since the substitution $x_v = R_v/b_v$ can always be done, we find that for *any* feasible batch-rate $x_v : v \in V$ in the explicit scheduling problem (8)–(10) there is an equivalent set of fractional buffer triggers $b_v : v \in V$ in the run-to-completion problem (11)–(13) and *vice versa*; put it another way, *any system state (collection of flow rates and delays) feasible under the explicit scheduling model is also attainable with a sufficient run-to-completion schedule*. To the best of our knowledge, this is the first time that an equivalence between the two crucial data flow-graph scheduling models is shown. We note however that the assumptions in Section 3.3 are critical for the equivalence to hold. For instance, explicit scheduling allows for a "lossy" schedule whereas a run-to-completion schedule is lossless by nature; under the "feasibility" assumption, however, there is no packet loss and hence the equivalence holds.

## 4 Implementation

We now describe the design and implementation of Batchy, our batch-scheduler for data flow graph packet-processing engines. The implementation follows the model introduced above, extended with a couple of heuristics to address practical limitations. We highlight only the main ideas of the implementation below; a detailed description of the heuristics with complete pseudocode can be found in the Appendix.

**Design.** To exploit the advantages of the simple architecture and zero scheduling overhead, in this paper we concentrate on the run-to-completion model (11)–(13) exclusively and we leave the implementation of the explicit scheduling model (8)–(10) for further study. This means, however, that currently we can enforce only delay-type SLOs using Batchy. In our design, the control plane constantly monitors the data plane and periodically intervenes to improve batch-efficiency and satisfy SLOs. Compared to a typical scheduler, Batchy interacts with the data plane at a coarser grain: instead of operating at the granularity of individual batches, it controls the pipeline by periodically adjusting the fractional buffer triggers and then it relies entirely on run-to-completion scheduling to handle the fine details of module execution.

**Receding-horizon control.** A naive approach to implement the control plane would be to repeatedly solve the convex program (11)–(13) and apply the resulting optimal fractional-buffer triggers to the data plane. Nevertheless, by the time the convex solver finishes computing the optimal schedule the system may have diverged substantially from the initial state with respect to which the solution was obtained. To tackle this difficulty, we chose a *one-step receding-horizon control* framework to implement Batchy. Here, in each control period the optimization problem (11)–(13) is bootstrapped with the current system state and fed into a convex solver, which is then is stopped *after the first iteration*. This results a coarse-grain control action, which is then immediately applied to the data plane. After the control period has passed, the system is re-initialized from the current state and a control is calculated with respect to this new state. This way, the controller rapidly drives the system towards improved states and eventually reaches optimality in steady state, automatically adapting to changes in the input parameters and robustly accounting for inaccuracies and failed model assumptions without having to wait for the convex solver to fully converge in each iteration.

**Main control loop.** Upon initialization, Batchy reads the data flow graph, the flows with the SLOs, and per-module service time profiles from the running system. During runtime, it measures in each control period the execution rate $\tilde{x}_v$, the packet rate $\tilde{r}_v$, and the mean batch size $\tilde{b}_v^{in}$ at the *input* of the ingress queue for each module $v \in V$, plus the 95th percentile packet delay $\tilde{t}_f$ measured at the egress of each flow $f \in \mathcal{F}$. (The overbar tilde notation is to distinguish *measured* parameters.) The statistics and the control period are configurable; e.g., Batchy can be easily re-configured to control for the 99th percentile or the mean delay. Due to its relative implementation simplicity and quick initial convergence, we use the gradient projection algorithm [5] to compute the control but in each run we execute only a *single* iteration. The algorithm will adjust triggers so as to obtain the largest relative gain in total system load and, whenever this would lead to an SLO violation, cut back the triggers just enough to avoid infeasibility.

**Insert/short-circuit buffers.** An unnecessary buffer on the packet-processing fast path introduces considerable delay and incurs nontrivial runtime overhead. In this context, a buffer is "unnecessary" if it already receives large enough batches at the input (Batchy detects such cases by testing for $\tilde{b}_v^{in} \geq 0.7B$); if it would further fragment batches instead of reconstructing them ($b_v \leq \tilde{b}_v^{in}$); or if just introducing the buffer already violates the delay-SLO ($1/x_v > D_f$ for some $v \in p_f$). If one of these conditions hold for a module $v$, Batchy immediately short-circuits the buffer in $v$ by setting the trigger to $b_v = 0$: the next-time module $v$ is executed the ingress queue is flushed and subsequent input batches are immediately fed into the network function without buffering. Similar heuristics allow Batchy to inject buffers into the running system: at initialization we short-circuit all buffers ("null-control", see below) and, during runtime, we install a buffer whenever *all* flows traversing a module provide sufficient delay-budget.

**Recovering from infeasibility.** The projected gradient controller cannot by itself recover from an infeasible (SLO-violating) state, which may occur due to packet rate fluctuation or an overly aggressive control action. A flow $f$ is in SLO-violation if $\tilde{t}_f \geq (1-\delta)D_f$ where $\delta$ is a configurable parameter that allows to trade off SLO-compliance for batch-efficiency. Below, we use the setting $\delta = 0.05$, which yields a rather aggressive control that strives to maximize batch size with a tendency to introduce relatively frequent, but small, delay violations. Whenever a flow $f \in \mathcal{F}$ is in SLO violation and there is a module $v$ in the path of $f$ set to a non-zero trigger ($b_v > \tilde{b}_v^{in}$), we attempt to reduce $b_v$ by $\left\lceil \frac{D_f - t_f}{\partial t_f / \partial b_v} \right\rceil$. If possible, this would cause $f$ to immediately recover from the SLO-violation. Otherwise, it is possible that the invariant $b_v \geq \tilde{b}_v^{in}$ may no longer hold; then we repeat this step at as many modules along $p_f$ as necessary and, if the flow is still in SLO-violation, we short-circuit all modules in $p_f$.

**Pipeline decomposition.** Batchy contains a pipeline controller responsible for migrating flows between workers to enforce otherwise unenforceable delay-SLOs. Consider the running example in Fig. 2, assume a single worker, let the processing cost of NF1 be 1 unit and that of NF2 be 10 units, and let the delay-SLO for the first flow be 2 units. This pipeline is inherently in delay-SLO violation: in the worst case a packet may need to spend 10 time units in the ingress queue until NF2 finishes execution, significantly violating the delay-SLO for the first flow (2 units). This inherent SLO violation will persist as long as NF1 and NF2 share a single worker. Batchy uses the analytical model to detect such cases: a worker $w$ is in inherent delay-SLO violation if there is a flow $f \in \mathcal{F}$ for which $\sum_{v \in V_w}(T_{v,0} + T_{v,1}B) \geq D_f$ holds, using the conservative estimate (3). Then Batchy starts a flow migration process: first it packs flows back to the original worker as long as the above condition is satisfied and then the rest of the flows are moved to a new worker. This is accomplished by decomposing the data flow graph into multiple disjunct connected worker subgraphs. Note that flows are visited in the ascending order of the delay-SLO, thus flows with restrictive delay requirements will stay at the original worker with a high probability, exempt from cross-core processing delays [19].

**Implementation.** We implemented Batchy on top of BESS [14] in roughly 6,000 lines of Python/BESS code. (Batchy is available at [4].) BESS is a high-performance packet processing engine providing a programming model and interface similar to Click [31]. BESS proved an ideal prototyping platform: it has a clean architecture, is fast [24], provides an efficient scheduler, exposes the right abstractions and offers a flexible plugin infrastructure to implement the missing ones. The distribution contains two built-in controllers. The *on-off controller* ("Batchy/on-off") is designed for the case when fractional buffers are not available in the data plane. This controller alters between two extremes (bang-bang control): at each module $v$, depending on the delay budget it either disables buffering completely ($b_v = 0$) or switches to full-batch buffering ($b_v = B$), using the above buffer-insertion/ deletion and feasibility-recovery heuristics. On top of this, the *full-fledged Batchy controller* ("Batchy/full") adds fractional buffers and fine-grained batch-size control using the projected gradient method.

## 5 Evaluation

Batchy is a control framework for packet-processing engines, which, depending on flows' offered packet rate and delay-SLOs, searches for a schedule that balances between batch-processing efficiency and packet delay. In this context the following questions naturally arise: *(i)* how much do batches fragment in a typical use case (if at all) and how much efficiency is there to gain by reconstructing these? how precisely does Batchy enforce SLOs?; *(ii)* which is the optimal operational regime for Batchy and what is the cost we pay?; *(iii)* does Batchy react quickly to changes in critical system parameters?; and finally *(iv)* can Batchy recover from inherent SLO-violations? Below we seek to answer these questions.

**Evaluation setup.** To understand the performance and latency-related impacts of batch control, we implemented two baseline controllers alongside the basic Batchy controllers (Batchy/on-off and Batchy/full): the *null-controller* performs no batch de-fragmentation at all ($b_v = 0 : v \in V$), while the *max-controller* reconstructs batches in full at the input of all modules ($b_v = B : v \in V$). Both baseline controllers ignore SLOs all together. The difference between performance and delay with the null- and max-controllers will represent the maximum attainable efficiency improvement batching may yield, and the price we pay in terms of delay. We also compared Batchy to NFVnice, a scheduling framework originally defined for the NFV context [21]. NFVnice is implemented in a low-performance container-based framework; to improve performance and to compare it head-to-head, we re-implemented its core functionality within BESS. Our implementation uses WFQ to schedule modules with equal weights and enables backpressure. All controllers run as a separate Python process, asserting *real-time control* over the BESS data plane via an asynchronous gRPC channel.

The evaluations are performed on 5 representative use cases taken from an official industry 5G benchmark suite [24]. The L2/L3($n$) pipeline implements a basic IP router, with L2 lookup, L3 longest-prefix matching, and group processing for $n$ next-hops; the GW($n$) use case extends this pipeline into a full-fledged gateway with NAT and ACL processing for $n$ next-hop groups; and the VRF($m$,$n$) pipeline implements $m$ virtual GW($n$) instances preceded by an ingress VLAN splitter (see Fig. 5). The MGW($m$,$n$) pipeline is a full 4G mobile gateway data plane with $m$ users and $n$ bearers per

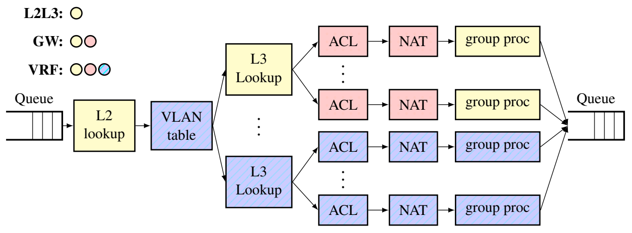


Figure 5: The VRF pipeline. The GW pipeline is identical to the VRF pipeline with only a single VRF instance, and the L2/L3 pipeline is a GW pipeline without a NAT and ACL on the next-hop branches.

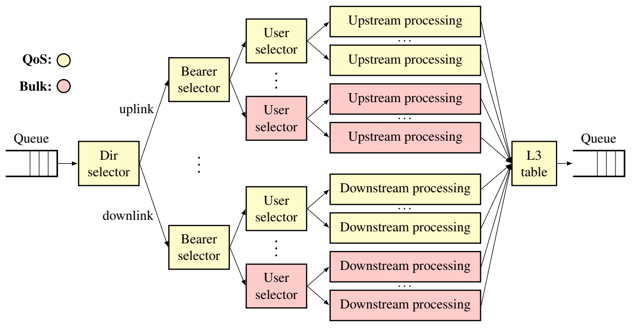


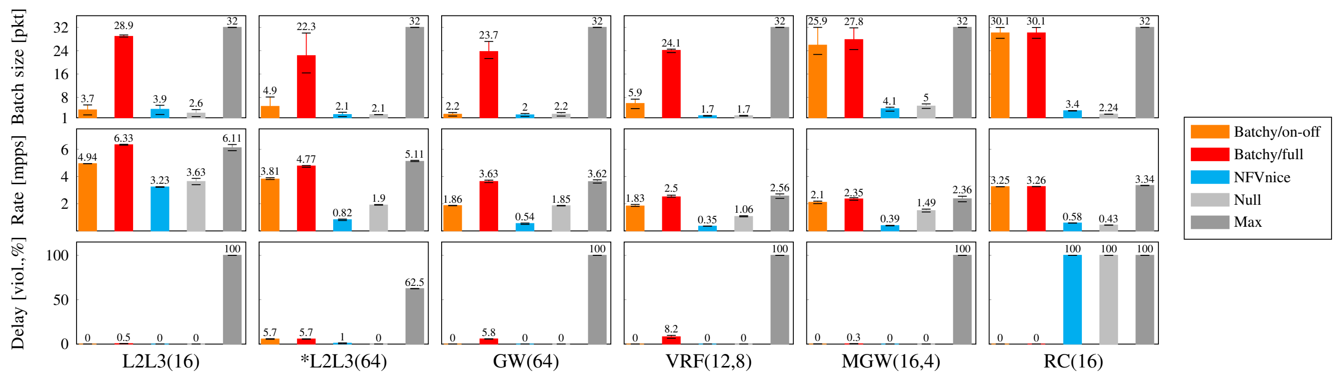
Figure 6: The mobile gateway (MGW) pipeline.

user, with complete uplink and downlink service chains (see Fig. 6). Finally, the RC($n$) pipeline models a 5G robot control use case: RC($n$) corresponds to the running example in Fig. 2 with $n$ branches, with the upper branch representing an ultra-delay-sensitive industry automation service chain and the rest of the branches carrying bulk traffic. We obtained test cases of configurable complexity by varying the parameters $m$ and $n$ and installing a flow over each branch of the resultant pipelines; e.g., in the VRF(2,4) test we have a separate flow for each VRF and each next-hop, which gives 8 flows in total.

Each pipeline comes with a traffic source that generates synthetic test traffic for the evaluations. For the L2/L3 pipeline, we repeated the tests with a real traffic trace taken from a CAIDA data set [9], containing 1.85 million individual transport sessions of size ranging from 64 bytes to 0.8 Gbytes (18 Kbyte mean) and maximum duration of 1 min (5.4 sec mean); the results are marked with the label *L2L3. Unless otherwise noted, the pipelines run on a single worker (single CPU core), with the traffic generator provisioned at another worker. The maximum batch size is 32, the control period is 100 msec, and results are averaged over 3 consecutive runs.

Each evaluation runs on a server equipped with an Intel Xeon E5-2620 v3 CPU (12 cores total, 6 isolated, power-saving disabled, single socket) with 64GB memory (with 12 $\times$ 1GB allocated as hugepages), installed with the Debian/ GNU operating system, Linux kernel v4.19, a forked version of BESS v0.4.0-57-g43bebd3, and DPDK v17.11.

**Batch-scheduling constant bit-rate flows.** In this test round, we benchmark the static performance of the Batchy con-



Common parameters. test time: 200 periods, warmup: 100 periods, control period: 0.1sec, $\delta = 0.05$, SLO-control: 95th percentile delay, burstiness: 1. Measure params (bucket/max): 5$\mu$sec/250msec. L2L3: FIB: 500 entries. CAIDA trace: `equinix-nyc.dirA.20190117-130600.UTC.anon`. GW: ACL: 100 entries, NAT: static NAT. MGW: FIB: 5k entries, 4 users on bearer0, upstream/downstream processing $T_{0,v} = 2000, T_{1,v} = 100$.

Figure 7: Static evaluation results (mean, 1st and 3rd quartile) with the null-controller, max-controller, NFVnice, Batchy/on-off and Batchy/full on different pipelines: average batch-size, cumulative packet rate, and delay-SLO statistics as the percentage of control periods when an SLO-violation was detected for at least one flow.

trollers against the baselines and NFVnice over 6 representative 5G NFV configurations. The null-controller, the max-controller, and NFVnice do not consider the delay-SLO, while for Batchy/on-off and Batchy/full we set the delay-SLO to 80% of the average delay measured with the max-controller; this setting leaves comfortable room to perform batch de-fragmentation but sets a firm upper bound on triggers. (Without an SLO, Batchy controllers degrade into a max-controller.)

After a warmup (100 control periods) we ran the pipeline for 100 control periods and we monitored the batch size statistics averaged across modules, the cumulative packet rate summed over all flows, and the delay-SLO statistics as the percentage of control periods when an SLO violation occurs for *at least* one flow. Fig. 7 highlights the results for 6 select configurations and Appendix B details the full result set. Our observations are as follows.

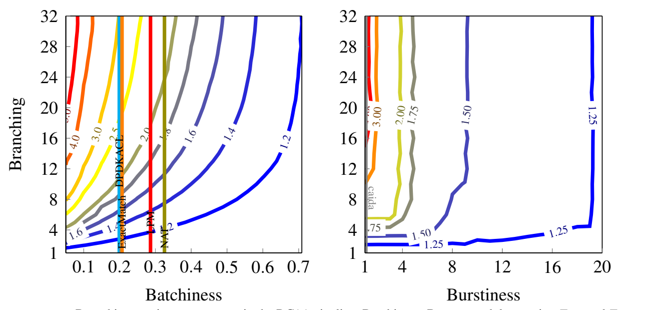
First, *the full-fledged Batchy controller (Batchy/full) can successfully reconstruct batches* at the input of network functions, achieving 70–80% of the average batch size of the max-controller in essentially all use cases (same proportion as the delay-SLO constraints). *Batch-fragmentation gets worse as the number of branches increases* across which batches may be split (the *branching* factor), to the point that when there are 16–64 pathways across the data flow graph the null-controller works with just 2–3 packets per batch. The simplified controller (Batchy/on-off) produces mixed results: whenever there is enough delay-budget to insert full buffers it attains similar de-fragmentation as Batchy/full (MGW(16,4), RC(16)), while in other cases it degrades into null-control (GW(64)).

Second, *batch de-fragmentation clearly transforms into considerable efficiency improvement*. Batchy/full exhibits 1.5–2.5$\times$ performance compared to the case when we do no batch de-fragmentation at all (null-control), and Batchy/on-off shows similar, although smaller, improvements. In the robot-control use case we see 7.5$\times$ throughput margin. This experiment demonstrates the benefits of selective per-module batch-control: there is only one highly delay-sensitive flow but this alone rules out any attempt to apply batching *globally* (even at the I/O); Batchy can, however, identify this chain and short-circuit all buffers along just this chain while it can still buffer the remaining bulk flows, yielding a dramatic performance boost. Despite the firm upper bound on the delay, and on the maximum attainable batch size, Batchy performs very close to the max-controller and in some tests even outperforms it (e.g., for L2L3(16)). This is because the max-controller buffers all modules unconditionally while Batchy carefully removes unnecessary buffers, and this helps in terms of valuable CPU cycles saved. The results are consistently reproduced over both the synthetic and the real traffic traces.

Third, despite the rather aggressive controller settings ($\delta = 0.05$, see the previous Section), *Batchy controllers violate the delay-SLO at most* 9% *of time* for at least one out of the possibly 64 flows, and even in these cases the relative delay violation is always below 1–2% (not shown in the figures). We believe that this is a price worth paying for the efficiency gain; manually repeating a failed test with a less aggressive control ($\delta = 0.2$) eliminated delay-SLO violations all together, at the cost of somewhat lower throughput.

Finally, we see that the Batchy/on-off controller is already useful in its own right in that it produces substantial batch-performance boost in certain configurations, but it hardly improves on null-control in others. It seems that discrete on-off control is too coarse-grained to exploit the full potential of batch de-fragmentation; to get full advantage we need finer-grained control over batch sizes, and the ensuing delay, using fractional buffers and the Batchy/full controller.

**Optimal operational regime and runtime overhead.** Next, we attempt to obtain a general understanding of the efficiency improvements attainable with Batchy, and the cost we pay in



Branching: setting parameter $n$ in the RC($n$) pipeline. Batchiness: Bypass module, varying $T_{0,v}$ and $T_{1,v}$.
Burstiness: varying mean burst size. Fixed parameters: first plot: burstiness=1; second plot: $\beta = 1/11$ for all modules.

Figure 8: Batchiness and burstiness vs. branching: Batchy/full packet rate normalized to the maximally fragmented case.

terms of controller overhead. For this, we first define a set of meta-parameters that abstract away the most important factors that shape the efficiency and overhead of Batchy, and then we conduct extensive evaluations in the configuration space of these meta-parameters.

The meta-parameters are as follows. Easily, it is the complexity of the underlying data flow graph that fundamentally determines Batchy's performance. We abstract pipeline complexity using the *branching* meta-parameter, which represents the number of distinct control-flow paths through the data flow graph; the higher the branching the more batches may break up inside the pipeline and the larger the potential batch de-fragmentation gain. Second, *batchiness*, as introduced in Section 2, determines each module's sensitivity to batch size; small batchiness usually indicates huge potential de-fragmentation gain. Finally, the specifics of the input traffic pattern is captured using the *burstiness* meta-parameter, which measures the average size of back-to-back packet bursts (or flowlets [37]) at the ingress of the pipeline. Indeed, burstiness critically limits batch-efficiency gains: as packet bursts tend to follow the same path via the graph they are less prone to fragmentation, suggesting that the performance margin of de-fragmentation may disappear over highly bursty traffic.

To understand how these factors shape the efficiency of Batchy, Fig. 8 shows two contour plots; the first one characterizes the speedup with Batchy/full compared to null-control in the branching–batchiness domain and the second one measures branching against burstiness. The plots clearly outline the optimal operational regime for Batchy: *as the number of branches grows beyond* 4–8 *and batchiness remains under* 0.5 *we see* 1.5–4× *speedup, with diminishing returns as the mean burst size grows beyond* 10. These gains persist with realistic exogenous parameters; batchiness for real modules is between 0.2–0.3 (see Fig. 3) and the CAIDA trace burstiness is only 1.13 (see the vertical indicators in the plots). But even for very bursty traffic and/or poor batch sensitivity, Batchy consistently brings over 1.2× improvement and never worsens performance: for workloads that do not benefit from batch de-fragmentation Batchy rapidly removes useless buffers and falls back to default, unbuffered forwarding.

| Branching ($n$) | Response time | Stats | Gradient | Control |
|---:|---:|---:|---:|---:|
| 1 | 2.4 msec | 66% | 31% | 3% |
| 2 | 3.5 msec | 61% | 37% | 2% |
| 4 | 6.9 msec | 65% | 32% | 3% |
| 8 | 11.6 msec | 65% | 32% | 3% |
| 16 | 21.9 msec | 66% | 30% | 4% |
| 32 | 34.8 msec | 68% | 28% | 4% |
| 64 | 89.1 msec | 72% | 22% | 6% |

Table 1: Batchy/full runtime overhead on increasingly more complex RC($n$) pipelines: branching, total controller response time, and contribution of each phase during the control, i.e., monitoring (Stats), gradient control (Gradient), and applying the control (Control).

Table 1 summarizes the controller runtime overhead in terms of the "pipeline complexity" meta-parameter (branching). The profiling results indicate that *the performance gains come at a modest resource footprint*: depending on pipeline complexity the controller response time varies between 2–90 milliseconds, with roughly two thirds of the execution time consumed by marshaling the statistics out from the data-plane and applying the new triggers back via gRPC, and only about one third taken by running the gradient controller itself.

**System dynamics under changing delay-SLOs.** We found the projected gradient controller to be very fast in the static tests: whenever the system is in steady state (offered load and delay-SLOs constant), Batchy usually reaches an optimal KKT point [5] in about 5–10 control periods. In the tests below, we evaluated Batchy under widely fluctuating system load to get a better understanding of the control dynamics.

First, we study how Batchy reacts to changing delay-SLOs (see the `conf/l2l3_vd.batchy` config file in the Batchy source distribution [4]). The results are in Fig. 9a. In this experiment, we set up the VRF(4, 8) pipeline and vary the delay-SLO between 60 $\mu$sec and 300 $\mu$sec in 6 steps; see the blue dotted "square wave" in Fig. 9a/Delay panel. The SLOs were set so that we test abrupt upwards ("rising edge") and downwards ("falling edge") delay-SLO changes as well. The figure shows for each control period the value of the trigger at the ACL module of the top branch (first VRF, first next-hop), the total delay and the delay-SLO for the flow provisioned at the top branch, and the normalized cumulative packet rate with the null-controller, the max-controller, and Batchy/full.

The results suggest that *Batchy promptly reacts to "bad news"* (SLO-reduction, falling edge, Fig. 9a/Delay) and instantaneously reduces fractional buffer triggers (Fig. 9a/Control), or even completely short-circuits buffers to recover from SLO-violations, whereas it is much more *careful to react to "good news"* (increasing SLO, rising edge). Overall, the packet delay closely tracks the SLO dynamics. Meanwhile, whenever there is room to perform batch de-fragmentation Batchy rapidly reaches the efficiency of the max-controller, *delivering* 2–3× *the total throughput of the null-controller*.

**System dynamics with variable bitrate traffic.** Next, we

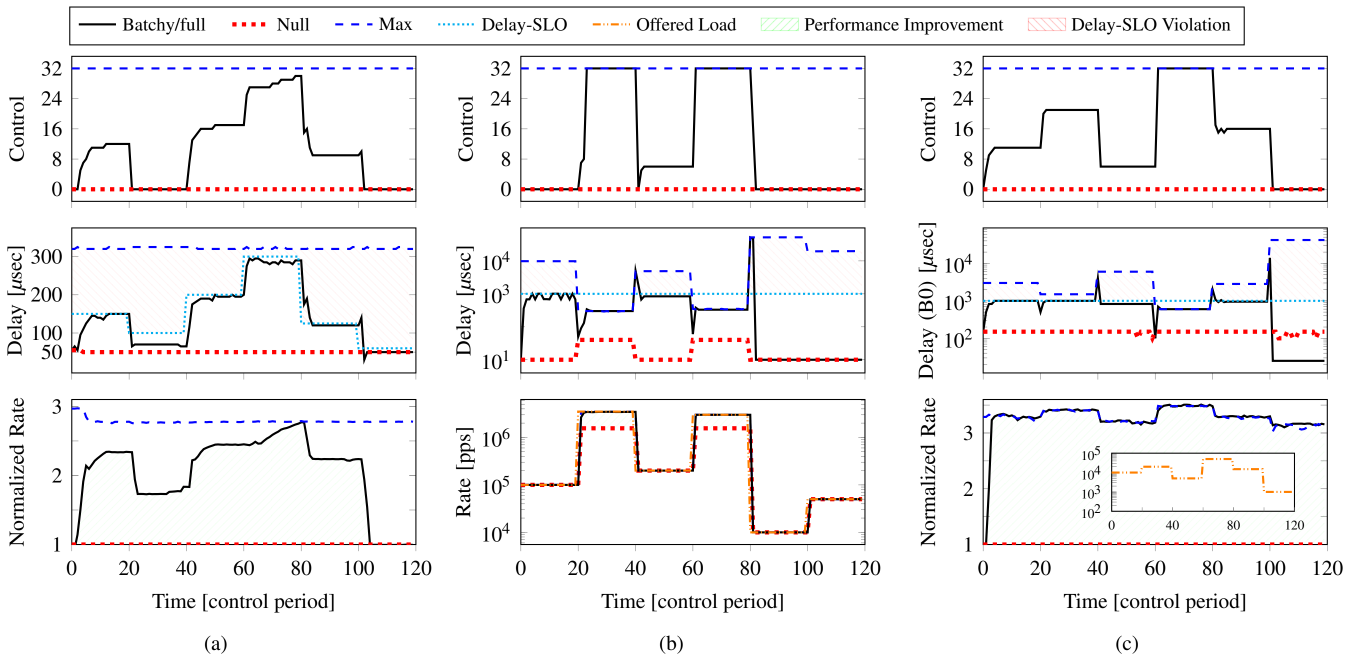


Figure 9: System dynamics with changing SLOs and variable bit-rate traffic: (a) control and delay for the first flow, and cumulative packet rate in the VRF(4,8) pipeline when the delay-SLO changes in the range 60–300 μsec; (b) control and delay for the first flow, and cumulative packet rate in the VRF(4,8) pipeline with the delay-SLO of all flows fixed at 1 msec and varying the total offered load between 50 kpps to 3 mpps; and (c) control, delay, and packet rate for the first user's bearer-0 (B0) flow in the MGW(2,16) pipeline, delay-SLO fixed at 1 msec, bearer-0 rate varying between 1 kpps and 50 kpps for all users.

test Batchy with variable bitrate flows. Intuitively, when the offered load drops abruptly it suddenly takes much longer for a buffer to accumulate enough packets to construct a batch, which causes delay to skyrocket and thereby leads to a grave delay-SLO violation. In such cases Batchy needs to react fast to recover. On the other hand, when the packet rate increases and queuing delays fall, Batchy should gradually increase triggers across the pipeline to improve batch-efficiency.

To understand the system dynamics under changing packet rates, we conducted two experiments. First, we fire up the VRF(4,8) pipeline and we fix the delay-SLO for all flows at 1 msec and vary the total offered load between 50 kpps to 3 mpps in 6 steps; this amounts to a dynamic range of 3 orders of magnitude (see `conf/l2l3_vbr.batchy` in [4]). Second, we set up the MGW(2,16) pipeline (2 bearers and 16 users, see `conf/mgw_vbr.batchy` in [4]), but now only bearer-0 flows (B0, the "QoS" bearer) vary the offered packet rate (between 1 kpps and 50 kpps) and set a delay-SLO (again, 1 msec). The results are in Fig. 9b and Fig. 9c, respectively.

Our observations here are similar as before. Even in the face of widely changing packet rates, Batchy keeps delay firmly below 1 msec except under transients: it *instantaneously recovers from SLO violations and rapidly returns to operate at full batch-size whenever possible*. Meanwhile, the max-controller causes a 100× SLO violation at small packet rates. In the second experiment we again see significant improvement in the total throughput with Batchy, compared to the null-controller (recall, only bearer-0 rate is fixed in this case).

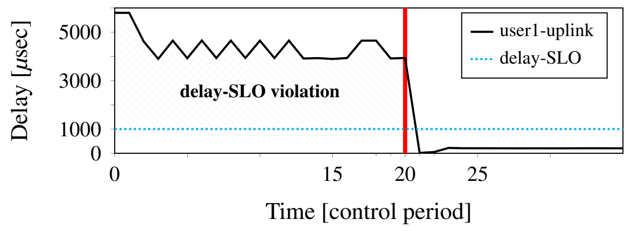


Figure 10: Recovery from inherent delay-SLO violations: MGW(2,8) test case with two users at bearer-0, delay-SLO set to 1 msec. The pipeline controller is started manually at the 20-th control period, moving bulk bearer-1 uplink and downlink traffic to a new worker each. (Downlink traffic and the other user's traffic exhibit similar performance.)

**Resource-(re)allocation when SLOs cannot be satisfied.** Finally, we study the effects of inherent delay-SLO violations, which occur when the turnaround time grows prohibitive at an over-provisioned worker and ingress queue latency exceeds the delay-SLO even before packets would enter the pipeline. Batchy implements a pipeline controller to detect inherent delay-SLO violations and to heuristically re-allocate resources, decomposing the data flow graph to multiple subgraphs to move delay-insensitive traffic to new workers.

Fig. 10 shows the pipeline controller in action (see `conf/mgw_decompose.batchy`) in [4]). Again we set up the

MGW(2,8) pipeline but now only two users open a flow at bearer-0, with delay-SLO set to 1 msec both in the uplink and downlink direction. The rest of the users generate bulk traffic at bearer-1 with no delay-SLO set. In addition, the service time of the bearer-1 uplink/downlink chains is artificially increased, which causes the turnaround time to surpass 1 msec and the latency for the delay-sensitive bearer-0 flows to jump to 4 msec. The pipeline controller kicks in at the 20-th control period and *quickly recovers the pipeline from the inherent delay-SLO violation*: by decomposing the data flow graph at the output of the `Bearer selector` splitter module, it moves all bearer-1 traffic away to new workers. The delay of bearer-0 flows quickly falls below the SLO, so much so that from this point Batchy can safely increase buffer sizes across the pipeline, leading to more than 10× improvement in the cumulative throughput (not shown in the figure).

## 6 Related Work

**Batch-processing in data-intensive applications.** Earlier work hints at the dramatic performance improvement batch-processing may bring in data-intensive applications and in software packet I/O in particular [1,6,7,20,29,41,50]. Consequently, batch-based packet-processing has become ubiquitous in software network switches [2,3,14,15,20,27,30,36], OS network stacks and dataplanes [3, 6, 8, 11], user-space I/O libraries [1, 16], and Network Function Virtualization [19,21,42,45,50]. Beyond the context of performance-centric network pipelines, batch-processing has also proved useful in congestion control [37], data streaming [18], analytics [44], and machine-learning [10].

**Dynamic batch control.** Clearly, the batch size should be set as high as possible to maximize performance [1,6,8,11,16,20, 27,29,41,50]; as long as I/O rates are in the range of multiple million packets per second the delay introduced this way may not be substantial [16]. Models to compute the optimal batch size *statically* and *globally* for the entire pipeline, subject to given delay-SLOs, can be found in [6,22,41,50]; [41] presents a discrete Markovian queuing model and [22] presents a discrete-time model for a single-queue single-server system (c.f. Fig. 4) with known service-time distribution. *Dynamic* batch-size control was proposed in [29], but again this work considers packet I/O only. Perhaps the closest to Batchy is IX [6], which combines run-to-completion and batch-size optimization to obtain a highly-efficient OS network data-plane, and NBA [20], which observes the importance avoiding "the batch split problem" in the context of data flow graph scheduling. Batchy extends previous work by providing a unique combination of dynamic *internal batch de-fragmentation* (instead of applying batching only to packet I/O), analytic techniques for *controlling queue backlogs* (using a new abstraction, fractional buffers), and *selective SLO-enforcement* at the granularity of individual service chains (extending batching to bulk flows even in the presence of delay-sensitive traffic).

**Data flow graph scheduling.** Data flow graphs are universal in data-intensive applications, like multimedia [43], machine learning [10], and robot control [39]. However, most of the previous work on graph scheduling considers a different context: in [23,33] the task is to find an optimal starting time for the parallel execution of processing nodes given dependency chains encoded as a graph, while [13] considers the version of the scheduling problem where graph-encoded dependencies exist on the input jobs rather than on the processing nodes. Neither of these works takes batch-processing into account.

**Service chains and delay-SLOs.** With network function virtualization [26] and programmable software switches [24,38] becoming mainstream, scheduling in packet-processing systems has received much attention lately. Previous work considers various aspects of NF scheduling, like parallel [42] implementation on top of process schedulers [21], commodity data-centers [19,35], and run-to-completion frameworks in an isolated manner [36], or in hybrid CPU–GPU systems [15,20,48,49]. Recent work also considers SLOs: [45] uses CPU cache isolation to solve the noisy neighbor problem while [50] extends NFV to GPU-accelerated systems and observes the importance of controlling batch-size to enforce delay-SLOs. Apart from complementing these works, Batchy also contributes to recent effort on network function performance profiling (BOLT [17]), efficient worker/CPU resource allocation for enforcing delay-SLOs (Shenango [34]), and avoiding cross-CPU issues in NF-scheduling (Metron [19]).

## 7 Conclusions

In this paper we introduce Batchy, the first general-purpose data flow graph scheduler that makes batch-based processing a first class citizen in delay-sensitive data-intensive applications. Using a novel batch-processing profile and an analytic performance modeling framework, Batchy balances batch-processing efficiency and latency and delivers strict SLO-compliance at the millisecond scale even at multiple millions of packets per seconds of throughput. As such, Batchy could be used as a central component in 5G mobile cores (e.g., the MGW use case) and industry-automation (e.g., the robot-controller use case) applications and latency-optimized network function virtualization (e.g., the VRF use case). It may also find use outside the networking context, as the batch-scheduler in streaming, analytics, machine learning, or multimedia and signal processing applications. In these applications, however, the default run-to-completion execution model adopted in Batchy may not provide sufficient workload isolation guarantees; future work therefore involves implementing a WFQ controller based on the model (8)–(10) to incorporate Batchy into an explicit process-scheduling model.

## References

[1] Advanced Networking Lab/KAIST. Packet I/O Engine. `https:`

//github.com/PacketShader/Packet-IO-Engine.

[2] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, December 2018.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, pages 5–16, 2015.

[4] Batchy. https://github.com/hsnlab/batchy.

[5] Mokhtar S Bazaraa, Hanif D Sherali, and Chitharanjan M Shetty. *Nonlinear programming: Theory and algorithms*. John Wiley & Sons, 2013.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX OSDI*, pages 49–65, 2014.

[7] Ankit Bhardwaj, Atul Shree, V. Bhargav Reddy, and Sorav Bansal. A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines. In *ACM APSys*, pages 26:1–26:7, 2017.

[8] Jesper Dangaard Brouer. Network stack challenges at increasing speeds. Linux Conf Au, Jan 2015.

[9] The CAIDA UCSD Anonymized Internet Traces - 2019. Available at http://www.caida.org/data/passive/passive_dataset.xml, 2019.

[10] Hong-Yunn Chen et al. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, volume 16, pages 265–283, 2016.

[11] Jonathan Corbet. Batch processing of network packets. Linux Weekly News, Aug 2018.

[12] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, pages 27–38, 2014.

[13] Mark Goldenberg, Paul Lu, and Jonathan Schaeffer. TrellisDAG: A system for structured DAG scheduling. In *JSSPP*, pages 21–43, 2003.

[14] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[16] Intel. Data Plane Development Kit. http://dpdk.org.

[17] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *USENIX NSDI*, pages 517–530, 2019.

[18] Apache Kafka. https://kafka.apache.org.

[19] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI*, pages 171–186, 2018.

[20] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A high-performance packet processing framework for heterogeneous processors. In *EuroSys*, pages 22:1–22:14, 2015.

[21] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM*, pages 71–84, 2017.

[22] S. Lange, L. Linguaglossa, S. Geissler, D. Rossi, and T. Zinner. Discrete-time modeling of NFV accelerators that exploit batched processing. In *IEEE INFOCOM*, pages 64–72, April 2019.

[23] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, June 1991.

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

[25] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *ACM SIGCOMM*, pages 1–14, 2016.

[26] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, pages 1–19, 2019.

[27] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed software data plane via vectorized packet processing. *Tech. Rep.*, 2017. https://perso.telecom-paristech.fr/drossi/paper/vpp-bench-techrep.pdf.

[28] Nesredin Mahmud et al. Evaluating industrial applicability of virtualization on a distributed multicore platform. In *IEEE ETFA*, 2014.

[29] M. Miao, W. Cheng, F. Ren, and J. Xie. Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing. In *IEEE HPCC*, pages 726–733, Dec 2016.

[30] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *ACM SIGCOMM*, pages 539–552, 2016.

[31] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *ACM SOSP*, pages 217–231, 1999.

[32] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, RFC Editor, January 1984.

[33] T. W. O'Neil, S. Tongsima, and E. H. Sha. Extended retiming: optimal scheduling via a graph-theoretical approach. In *IEEE ICASSP*, volume 4, 1999.

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX NSDI*, pages 361–378, 2019.

[35] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *ACM SOSP*, pages 121–136, 2015.

[36] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX OSDI*, pages 203–216, 2016.

[37] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *USENIX NSDI*, pages 421–435, 2017.

[38] Ben Pfaff et al. The design and implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.

[39] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[40] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *USENIX NSDI*, pages 33–46, February 2019.

[41] Z. Su, T. Begin, and B. Baynat. Towards including batch services in models for DPDK-based virtual switches. In *GIIS*, pages 37–44, 2017.

[42] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: enabling network function parallelism in NFV. In *ACM SIGCOMM*, pages 43–56, 2017.

[43] Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost. *GStreamer Application Development Manual*. Samurai Media Limited, United Kingdom, 2016.

[44] The Apache Spark project. Setting the Right Batch Interval. `https://spark.apache.org/docs/latest/streaming-programming-guide.html#setting-the-right-batch-interval`.

[45] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *USENIX NSDI*, pages 283–297, 2018.

[46] Carl A Waldspurger and E Weihl W. Stride scheduling: deterministic proportional-share resource management, 1995. Massachusetts Institute of Technology.

[47] Ed Warnicke. Vector Packet Processing - One Terabit Router, July 2017.

[48] Xiaodong Yi, Jingpu Duan, and Chuan Wu. GPUNFV: A GPU-accelerated NFV system. In *ACM APNet*, pages 85–91, 2017.

[49] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU sharing in NFV systems. In *USENIX NSDI*, pages 187–200, 2018.

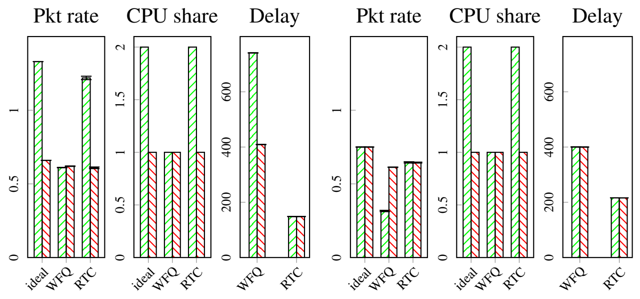
[50] Zhilong Zheng, Jun Bi, Haiping Wang, Chen Sun, Heng Yu, Hongxin Hu, Kai Gao, and Jianping Wu. Grus: Enabling latency SLOs for GPU-accelerated NFV systems. In *IEEE ICNP*, pages 154–164, 2018.

## Appendix

## A Data Flow Graph Scheduling

Scheduling in the context of data flow graphs means to decide which module to execute next. The goal of the scheduler is to provide efficiency and fairness: efficiency is ultimately determined by the amount of load the system can process from the ingress modules to the egress modules and fairness is generally measured by the extent to which the eventual resource allocation is *rate-proportional* [21]. Here, limited CPU resources need to be allocated between competing modules based on the combination of the offered load (or arrival rate) for the module and its processing cost. Intuitively, if either one of these metrics is fixed then the CPU allocation should be proportional to the other metric. Consider the example in Fig. 2; if the two modules have the same CPU cost but NF1 has twice the offered load than NF2, then we want it to have twice the CPU time allocated, and hence twice the output rate, relative to NF2. Alternatively, if the NFs have the same offered load but NF1 incurs twice the processing cost then we expect it to get twice as much CPU time, resulting in both modules having roughly the same output rate.

**Explicit scheduling.** In explicit scheduling there is a standalone mechanism that runs side-by-side with the pipeline and executes modules in the given order. A typical example is Weighted Fair Queueing (WFQ) or Completely Fair Scheduling (CFS), where the user assigns integer *weights* to



(a) Asymmetric rate (b) Asymmetric cost

Figure 11: Rate-proportional fairness in WFQ and run-to-completion scheduling in the *asymmetric rate* case (NF1 receives twice the offered load of NF2 and CPU costs are equal) and *asymmetric cost* case (same offered load but NF1 needs twice as much CPU time to process a packet as NF2). Packet rate is in mpps and delay is in $\mu$sec, and ▨ denotes the first flow while ▨ denotes the second flow as in Fig. 2.

| | | Batch size [pkt] | | | | | Rate [Mpps] | | | | | Delay [% of violations] | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Pipeline** | **# modules** | Null | Max | NFVnice | Batchy/on-off | Batchy/full | Null | Max | NFVnice | Batchy/on-off | Batchy/full | Null | Max | NFVnice | Batchy/on-off | Batchy/full |
| L2L3(1) | 9 | 31.5 | 32 | 31.8 | 31.5 | 31.5 | 8.29 | 8.31 | 4.78 | 8.29 | 8.47 | 0% | 100% | 100% | 0% | 0% |
| L2L3(4) | 21 | 12.6 | 32 | 14.53 | 12.6 | 29.2 | 7.49 | 7.46 | 5.86 | 7.26 | 7.77 | 0% | 100% | 13% | 0% | 0% |
| L2L3(8) | 37 | 7.0 | 32 | 8.1 | 5.8 | 30.9 | 6.26 | 7.38 | 4.65 | 5.18 | 7.42 | 0% | 100% | 0% | 0% | 0% |
| L2L3(16) | 69 | 2.6 | 32 | 3.92 | 3.7 | 28.9 | 3.63 | 6.11 | 2.75 | 4.94 | 6.33 | 0% | 100% | 0% | 0% | 0.5% |
| *L2L3(16) | 69 | 8.5 | 32 | 4.7 | 9.2 | 24.2 | 5.24 | 5.7 | 2.4 | 5.25 | 5.79 | 0% | 75% | 2% | 8.3% | 2.4% |
| *L2L3(32) | 133 | 3.1 | 32 | 3 | 5.8 | 24.5 | 4.47 | 5.32 | 1.46 | 4.61 | 5.48 | 0% | 63% | 0% | 1.9% | 1.3% |
| *L2L3(64) | 261 | 1.1 | 32 | 2.1 | 4.9 | 23.4 | 2.12 | 5.17 | 0.82 | 3.81 | 5.03 | 0% | 68% | 1% | 5.7% | 5.2% |
| GW(1) | 12 | 31.5 | 32 | 31.7 | 31.5 | 31.5 | 6.06 | 6.08 | 3.11 | 6.08 | 6.08 | 0% | 100% | 100% | 0% | 0% |
| GW(8) | 61 | 5.6 | 32 | 6.21 | 5.6 | 31.4 | 4.2 | 5.22 | 2.7 | 4.23 | 5.25 | 0% | 100% | 23.6% | 0% | 0.6% |
| GW(16) | 117 | 2.9 | 32 | 3.16 | 2.9 | 30.7 | 2.66 | 4.51 | 1.47 | 2.8 | 4.58 | 0% | 100% | 44.3% | 0% | 0.5% |
| GW(64) | 453 | 2.2 | 32 | 2 | 2.2 | 24.4 | 1.85 | 3.39 | 0.54 | 1.86 | 3.73 | 0% | 100% | 0% | 0% | 5.2% |
| VRF(1,1) | 11 | 32 | 32 | 32 | 32 | 32 | 5.88 | 5.76 | 3.48 | 5.87 | 5.92 | 0% | 100% | 100% | 0% | 0% |
| VRF(2,4) | 43 | 5.3 | 32 | 7.1 | 10.7 | 22.2 | 3.49 | 4.77 | 2.53 | 4.37 | 4.03 | 0% | 100% | 54% | 0% | 2.2% |
| VRF(16,4) | 323 | 2 | 32 | 2 | 10.8 | 24.4 | 1.01 | 2.7 | 0.42 | 2.41 | 2.84 | 0% | 100% | 0% | 0% | 8.9% |
| VRF(12,8) | 435 | 1.7 | 32 | 1.7 | 5.9 | 23.4 | 0.95 | 2.34 | 0.35 | 1.82 | 2.5 | 0% | 100% | 0% | 0% | 8.2% |
| MGW(2,4) | 110 | 4.5 | 32 | 4.5 | 13.7 | 17.3 | 1.75 | 3.15 | 1.03 | 1.78 | 2.03 | 0% | 100% | 100% | 0% | 0% |
| MGW(4,4) | 206 | 4 | 32 | 3.7 | 11.4 | 20.2 | 1.5 | 2.83 | 0.8 | 1.69 | 2.19 | 0% | 100% | 61.6% | 0% | 1% |
| MGW(8,4) | 398 | 5.1 | 32 | 4.1 | 20.9 | 25.7 | 1.61 | 2.77 | 0.6 | 2.12 | 2.5 | 0% | 100% | 0% | 0% | 5% |
| MGW(16,4) | 782 | 4.9 | 32 | 4.1 | 25.9 | 27.8 | 1.49 | 2.34 | 0.39 | 2.1 | 2.25 | 0% | 100% | 0% | 0% | 0.3% |
| RC(16) | 25 | 2.24 | 32 | 3.43 | 30.14 | 30.14 | 0.43 | 3.27 | 0.58 | 3.25 | 3.27 | 100% | 100% | 100% | 0% | 0% |

Table 2: Static evaluation results with the null-controller, max-controller, NFVnice, Batchy/on-off and Batchy/full on different pipelines: number of modules, average batch-size over the pipeline, packet rate, and delay statistics in terms of the percentage of control periods when a delay-SLO violation was detected for at least one flow.

modules and the scheduler ensures that the runtime resource allocation will be proportional to modules' weight. WFQ does not provide rate-proportional fairness out of the box; e.g., in the example of Fig. 2 NF1 will *not* receive more CPU time neither when its offered load (asymmetric rate) or processing cost (asymmetric cost) is twice that of NF2 (see Fig. 11). Correspondingly, WFQ schedulers need substantial tweaking to approximate rate-proportional fairness, and need further optimization to avoid head-of-line blocking and late drops along a service chain [21]. Even running the scheduler itself may incur non-trivial runtime overhead. Worse still, packets may get dropped *inside* the pipeline when internal queues overflow; this may be a feature (e.g., when we want to apply rate-limitation or traffic policing via the scheduler) or a bug (when useful traffic gets lost at an under-provisioned queue).

**Run-to-completion execution.** This model eliminates the explicit scheduler and its runtime overhead all together. In run-to-completion execution the entire input batch is traced though the data flow graph in one shot, by upstream modules automatically scheduling downstream modules whenever there is work to be done [6, 14]. As Fig. 11 shows, this model introduces much smaller delay vs. explicit scheduling, as it needs no internal queues. In addition, run-to-completion provides rate-proportional fairness out-of-the-box, even without additional tweaking and without the risk of head-of-line blocking and internal packet drops. This yields an appealingly simple "schedulerless" design. On the other hand, since module execution order is automatically fixed by the pipeline and the scheduler cannot by itself drop packets, the share of CPU time a module gets is determined by the offered load only. This makes enforcing rate-type SLOs through a run-to-completion scheduler difficult. In our example, if NF2 receives twice the packet rate of NF1 then it will receive twice the CPU share, and hence the second flow will have twice the output rate, even though we may want this to be the other way around.

## B Static Evaluations: Detailed Results

The detailed static performance results are given in Table 2. The table specifies the number of modules in each pipeline, the batch size statistics averaged across each module in time, the throughput as the cumulative packet rate summed over all flows, and the delay-SLO violation statistics as the percentage of control periods when we detected an SLO violation for at least one flow, for each of the 5G NF use cases, varying pipeline complexity using different settings for the parameters $n$ and $m$.

## C The Fractional Buffer

The below algorithm summarizes the execution model of a fractional buffer. Here, *queue* means the internal queue of the fractional buffer, $b$ is the trigger, *q.pop(x)* dequeues $x$ packets from the beginning of queue $q$, and $q.push(batch)$ appends the packets of $batch$ to the queue $q$.

```
procedure FRACTIONALBUFFER::PUSH($batch$)
    while $queue.size \leq b$ AND $batch.size > 0$ do
        $queue.push(batch.pop(1))$
    end while
    if $queue.size = b$ then
        $new\_batch \leftarrow queue.pop(b)$
        process $new\_batch$ through the network function
        put $v$'s downstream modules to the run queue
        $queue.push(batch.pop(batch.size))$
    end if
end procedure
```

## D The Projected Gradient Controller

Below, we discuss the high-level ideas in the projected gradient controller implemented in Batchy and then we give the detailed pseudocode.

First, we compute the objective function gradient $\nabla l =$

$[\partial l / \partial b_v : v \in V]$, which measures the sensitivity of the total system load $l = \sum_{v \in V} l_v$ as of (11) to small changes in the trigger $b_v$ for each module:

$$\frac{\partial l}{\partial b_v} = -\frac{\tilde{r}_v T_{0,v}}{b_v^2} = -\frac{\tilde{x}_v T_{0,v}}{b_v} \quad .$$

The delay-gradients $\nabla t_f = [\partial t_f / \partial b_v : f \in \mathcal{F}]$ are as follows:

$$\frac{\partial t_f}{\partial b_v} = \begin{cases} \frac{1}{\tilde{r}_v} + T_{1,v} & \text{if } v \in p_f \\ 0 & \text{otherwise} \end{cases}$$

Note that the delay $t_f$ of a flow $f$ is affected only by the modules along its path $p_f$, as long as the turnaround time is considered constant as of (3).

Second, project the objective gradient $\nabla l$ to the feasible (i.e., SLO-compliant) space. For this, identify the flows $f$ that may be in violation of the delay-SLO: $\tilde{t}_f \geq (1-\delta) D_f$.

Third, let $M$ be a matrix with row $i$ set to $\nabla t_f$ if $f$ is the $i$-th flow in delay violation and compute the projected gradient $\Delta b = -(I - M^T (MM^T)^{-1} M) \nabla l$. Note that if $M$ is singular or the projected gradient becomes zero then small adjustments need to be made to the projection matrix [5].

Fourth, perform a line-search along the projected gradient $\Delta b$. If for some module $v$ the corresponding projected gradient component $\Delta b_v$ is strictly positive (it cannot be negative) then calculate the largest possible change in $b_v$ that still satisfies the delay-SLO of *all* flows traversing $v$:

$$\lambda_v = \min_{f \in \mathcal{F} : v \in p_f} \frac{D_f - \tilde{t}_f}{\Delta b_v} \quad .$$

Finally, take $\lambda = \min_{v \in V} \lambda_v$ and adjust the trigger of each module $v$ to $b_v + \lceil \Delta b_v \lambda \rceil$. Rounding the trigger up to the nearest integer yields a more aggressive control.

The below pseudo-code describes the projected gradient controller in detail. Vectors and matrices are typeset in bold in order to simplify the distinction from scalars. We generally substitute matrix inverses with the Moore-Penrose inverse in order to take care of the cases when **M** is singular.

```
procedure ProjectedGradient($\mathcal{G}, \mathcal{F}, D, f$)
    $\mathbf{M}$ is a matrix with row $i$ set to $\nabla t_f = \left[\frac{\partial t_f}{\partial b_v} : f \in \mathcal{F}\right]$, where $f$
is the $i$-th flow in $\mathcal{F}$ with $\tilde{t}_f \geq (1-\delta) D_f$
    ▷ Gradient projection
    while True do
        $\mathbf{P} = \mathbf{I} - \mathbf{M}^T (\mathbf{M}\mathbf{M}^T)^{-1} \mathbf{M}$
        $\Delta \mathbf{b} = \mathbf{P} \nabla \mathbf{l}$
        if $\Delta \mathbf{b} \neq 0$ then break
        $\mathbf{w} = -(\mathbf{M}\mathbf{M}^T)^{-1} \mathbf{M} \nabla \mathbf{l}$
        if $\mathbf{w} \geq 0$ then return        ▷ Optimal KKT point reached
        delete row for $f$ from $\mathbf{M}$ for some $f \in \mathcal{F} : w_f < 0$
    end while
    ▷ Line search
    for $v \in V, f \in p_v$ do
        if $\Delta b_v > 0$ then
            $\lambda_v = \min_{f \in \mathcal{F} : v \in p_f} \left\lceil \frac{D_f - \tilde{t}_f}{\Delta b_v} \right\rceil$
        end if
    end for
    $\lambda = \min_{v \in V} \lambda_v$
    for $v \in V$ do SetTrigger($v$, $b_v + \Delta b_v \lambda$)
end procedure
```

## E The Feasibility-recovery Algorithm

The projected gradient controller cannot by itself recover from situations when a fractional buffer at some module is triggered at a too small rate to deliver the required delay-SLO to each flow traversing the module. The below pseudo-code describes the feasibility recovery process implemented in Batchy, which is implemented to handle such situations.

```
procedure FeasibilityRecovery($\mathcal{G}, \mathcal{F}, D, f$)
    for $f \in \mathcal{F}$ do $t_f \leftarrow \tilde{t}_f$
    ▷ Recover from SLO violation
    for $v \in V : b_v \geq \tilde{b}_v^{in}$ do
        if $\exists f \in \mathcal{F} : v \in p_f$ AND $t_f \geq (1-\varepsilon) D_f$ then
            $\Delta b_v = \max_{f \in \mathcal{F} : v \in p_f \wedge t_f \geq (1-\varepsilon) D_f} \left\lceil \frac{D_f - t_f}{\partial t_f / \partial b_v} \right\rceil$
            if $\Delta b_v > b_v - \tilde{b}_v^{in}$ then $\Delta b_v \leftarrow b_v - \tilde{b}_v^{in}$
            for $f \in \mathcal{F} : v \in p_f$ do $t_f \leftarrow t_f - \frac{\partial t_f}{\partial b_v} \Delta b_v$
            $b_v \leftarrow b_v - \Delta b_v$
        end if
    end for
    ▷ Injecting a buffer
    for $v \in V : b_v = 0$ do
        if $\forall f \in \mathcal{F} : v \in p_f$ it holds that $t_f < (1-\varepsilon) D_f$ then
            $\Delta b_v = \min_{f \in \mathcal{F} : v \in p_f \wedge t_f < (1-\varepsilon) D_f} \left\lceil \frac{t_f - D_f}{\partial t_f / \partial b_v} \right\rceil$
            if $\Delta b_v > \tilde{b}_v^{in}$ then $\Delta b_v \leftarrow \tilde{b}_v^{in}$
            for $f \in \mathcal{F} : v \in p_f$ do $t_f \leftarrow t_f + \frac{\partial t_f}{\partial b_v} \Delta b_v$
            $b_v \leftarrow \Delta b_v$
        end if
    end for
    for $v \in V$ do SetTrigger($v$, $b_v$)
end procedure
```

## F The Data Flow Graph Decomposition Algorithm

Pipeline decomposition is initiated in Batchy whenever an inherent delay-SLO violation is detected. This occurs when a worker is overprovisioned and the turnaround time grows beyond the SLO for a delay-sensitive flow; in such cases Batchy migrates delay-insensitive traffic to new workers to address SLO violations. The below pseudo-code describes the pipeline decomposition procedure implemented in Batchy.

```
procedure DecomposePipeline($\mathcal{G}, \mathcal{F}, D, f$)
    $V_t \leftarrow \emptyset$
    $F_t \leftarrow \emptyset$
    $\tau_t \leftarrow \emptyset$
    for $f \in$ ($F$ in ascending order of $D_f$) do
```

```
        **if** CHECK_DELAY_SLO($V_t, F_t, \tau_t, f$) **then**
            $F_t \leftarrow F_t \cup f$
            $V_t \leftarrow V_t \cup p_f$
            $\tau_t \leftarrow \tau_t + \sum_{v \in P_f, v \notin V_t}(T_{0v} + T_{1v}B)$
        **else**
            $M \leftarrow \emptyset$
            MIGRATEFLOWS($V_t, F_t, M, \mathcal{F}$)
        **end if**
    **end for**
**end procedure**
**procedure** MIGRATEFLOWS($V_t, F_t, M, \mathcal{F}$)
    **for** $g \in (\mathcal{F} \setminus F_t)$ **do**
        **for** $v \in p_g$ **do**
            **if** $v \notin V_t$ and $v \notin M$ **then**
                create new worker
                add a queue before $v$
                attach queue to new worker
                $M \leftarrow M \cup v$
            **end if**
        **end for**
    **end for**
**end procedure**
**procedure** CHECKDELAYSLO($V_t, F_t, \tau_t, f$)
    **for** $g \in (F_t \cup f)$ **do**
        **if** $(\tau_t + \sum_{v \in P_f, v \notin V_t}(T_{0v} + T_{1v}B) + \sum_{v \in P_g}(T_{0v} + T_{1v})) > D_g$
**then**
            **return** False
        **end if**
        **return** True
    **end for**
**end procedure**
```