Batch-scheduling Data Flow Graphs with Service-level Objectives on Multicore Systems

Tamás Lévai and Gábor Rétvári, *Member, IEEE*

***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. Batchy is a scheduler for data flow graph based packet processing engines, which uses controlled queuing to reconstruct fragmented batches inside a data flow graph in accordance with strict Service-Level Objectives (SLOs). Earlier work showed that Batchy yields up to 10x performance improvement in real-life use cases, thanks to maximally exploiting batch processing gains.**

Batchy, however, is fundamentally restricted to single-threaded execution. In this paper, we generalize Batchy to parallel execution on multiple CPU cores. We extend the analytical model to the parallel setting and present a primal decomposition framework, where each core runs an unmodified Batchy controller to schedule batch-processing on a subset of the data flow graph, orchestrated by a master controller that distributes the delay-SLOs across the cores using subgradient search. Evaluations on a real software switch provide experimental evidence that our decomposition framework produces 2.5x performance improvement while accurately satisfying delay SLOs that are otherwise not feasible with single-core Batchy.

***Index Terms*—data flow graph, decomposition, software switch, SDN, NFV**

I. Introduction

BATCH-SCHEDULING is a near-universal technique to improve performance of software packet processing engines: 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 [1], which often yields 2–5× performance boost. Consequently, batching is used in essentially all software switches (*e.g.,* BESS [2], VPP [3], FastClick [4], and ESwitch [5]), high-performance OS network stacks and libraries [6], user-space I/O libraries [7], and Network Function Virtualization (NFV) platforms [8], [9].

Batchy [10] is a state-of-the-art batch-scheduling framework for high-end programmable software switches. Batchy abstracts the software switch dataplane as a data flow graph; here, nodes represent packet-processing primitives (*e.g.,* L3 Lookup) and arcs represent the control flow. This data flow graph is executed in a *run-to-completion* fashion; when a packet-processing node finishes work on a packet batch, execution proceeds on the downstream nodes along all outgoing arcs of the node. Unfortunately, run-to-completion tends to fragment batches inside the data-flow graph, as each node may split the input batch into multiple sub-batches to be passed to downstream nodes; *e.g.,* an L3 Lookup table or a round-robin LoadBalancer may distribute the packets inside the batch across multiple downstream processing chains, a network stack may split a burst of mixed input packets per L3/L4 protocol to execute each MPLS, IPv4 and IPv6 packet on a separate downstream protocol engine, *etc.* Since the downstream modules are executed on smaller batches we lose batch-efficiency, which inherently curtails the available performance, often an order of magnitude lower than with full batches [1].

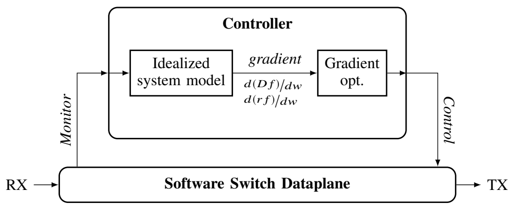

Figure 1. Batchy System Architecture.

Batchy attempts to recover some of the lost batch-efficiency by artificially queuing up packets inside the data flow graph to be able to execute the downstream processing nodes on larger batches. Inspired by Nagle's algorithm [11], Batchy uses a model-predictive controller to regulate queue backlogs for maximizing batch sizes across the pipeline in a way so that the end-to-end queuing delay remains under a given requirement (Fig. 1). This brings massive performance improvement, and delay Service Level Objective (SLO) conformance in the μs range even at million-packet-per-second scale traffic [10]. Unfortunately, the model underlying Batchy assumes single-core execution.

Motivated by the need to run software switches on multicore systems to maximize performance [12], [13], *in this paper we extend Batchy to leverage parallel execution.* As Fig. 2 shows, this is not trivial. The task is two-fold: *i)* find an optimal batch-schedule on each core, and *ii)* distribute delay budgets among cores in a way so that the end-to-end delay remains under the SLO. This is a two-level optimization problem: on per core basis the goal is to find the optimal queue backlog sizes and on a higher level to determine how long each core can process a packet batch so to meet end-to-end delay SLOs. To solve this complex multi-level problem, we propose a decomposition technique [14].

The general idea of decomposition is to break a complex problem into simpler subproblems, then solve the simple

T. Lévai is with the Department of Telecommunications and Media Informatics at the Budapest University of Technology and Economics. G. Rétvári is with the MTA-BME Information Systems Research Group and Ericsson Research. This work was supported by the NKFIH/OTKA Project #135606. E-mail: {levait, retvari}@tmit.bme.hu.

DOI: 10.36244/ICJ.2022.1.6

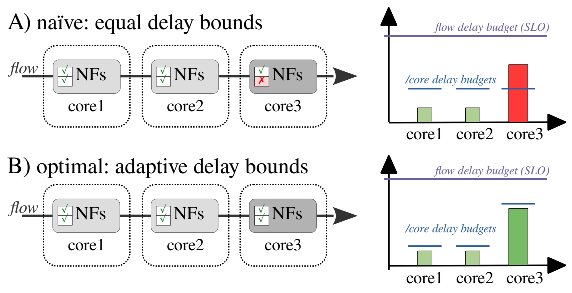

Figure 2. Motivating example for multicore Batchy [10]. The pipeline runs on 3 cores and serves a single flow. NFs on each core require a given amount of time to process a full packet batch (core1: 1, core2: 1, and core3: 4 units). Note that per-core delays add up, so that a flow's end-to-end delay equals the sum of the delay imposed on the flow's packets at each core. *a)* Naïve approach: no coordination between the CPU cores. This yields limited performance since the delay on core3 always exceeds the per-core delay budget and hence there is no room to reconstruct batches. *b)* Optimal adaptive per-core delay budget distribution: core3 now gets a higher delay budget than the rest of the cores. Per-core delay budgets are now satisfied and there is enough delay budget to efficiently defragment batches on core3, which then yields significant performance improvement.

subproblems separately under the control of a global problem that takes care of the "complicating constraints". This technique was already adapted to many networking domains, such as network utility maximization [15], radio transceiver design [16], and beamforming [17]. The goal of decomposition in Batchy is to split the global scheduling problem among the cores (*i.e.,* CPUs) in a multicore system, so that each core autonomously optimizes batch sizes across a subset of the data flow graph subject to a per-core flow delay budget, with minimal switch-level orchestration that adjusts the delay budgets per each core to meet the global delay SLOs. The per-core controller will be conveniently implemented by the unmodified single-core Batchy algorithm. This setup reflects a *primal decomposition* [14] structure.

Our contributions in this paper are as follows:

Analytical model. After a short recap [1] on Batchy (§II), we introduce an expressive mathematical model for SLO-based batch-scheduling on multicore software switches (§III). Our framework allows to formally reason about the performance and adaptively distribute end-to-end delay SLOs across cores to maximize performance.

Control algorithms. We design control algorithms for effective multicore batch-scheduling under delay SLOs (§IV).

Design, implementation, and evaluation. We present a practical implementation of the multicore scheduling framework by extending Batchy and using the BESS software switch [2] (see §IV). We demonstrate the effectiveness of our control algorithms in a realistic use case, VRF (Virtual Routing Function), taken from an official industry 5G NFV benchmarking suite [13]. We show that our control algorithms increase total packet rate by up to 2.5× beyond what is available with single-core Batchy, while meeting delay SLO requirements that are otherwise not feasible with single-core Batchy. Our implementation is available for download at [18].

We close the paper discussing related work (§VI) and deriving the main conclusions (§VII).

[1]In this paper we only introduce Batchy essentials due to space constraints. For Batchy details, we kindly refer the reader to the Batchy paper [10].

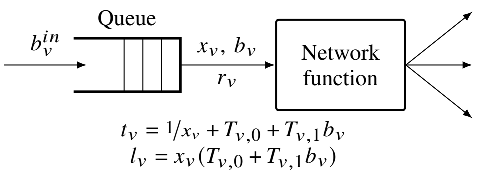
Queue
b_v^{in}
$x_v,\ b_v$
r_v
Network function
$t_v = 1/x_v + T_{v,0} + T_{v,1} b_v$
$l_v = x_v (T_{v,0} + T_{v,1} b_v)$

Figure 3. A Batchy Module.

II. Batchy System Model

Next, we introduce our analytical model. We mostly reproduce the main ideas from the single-core setting, highlighting the extensions we introduce for the multicore setting.

A. Concepts

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. 3). *Input gates* (or ingates) are represented as in-arcs $(u, v) \in E : u \in V$ and *output gates* (or outgates) as out-arcs $(v, u) \in E : u \in V$. 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 a module (*e.g.,* access control) would drop a packet it will rather send it to a dedicated "drop" gate, so that we can account for lost packets.

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], [5], [7]. 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, while GPU/NIC offload often works with $B = 1024$ or even larger to maximize I/O efficiency [8], [19].

Module service time profile. After extensive evaluation of network functions on various software switches, 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. Accordingly, we shall use the linear approximation $T_v = T_{v,0} + T_{v,1} b_v$ [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.

Module types. Any module may have multiple ingates (merger) and/or multiple outgates (splitter), or may have no ingate or outgate at all. An L3 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*. Similarly, a module with no outgates (*e.g.,* a transmit queue) is an *egress module*.

Compute resources. A *task* ($t \in \mathcal{T}$) is our main compute resource abstraction. Tasks are modeled as a connected subgraph $\mathcal{G}_t = (V_t, E_t)$ of $\mathcal{G}$, with strictly one ingress module representing an ingress queue that buffers packets between subsequent executions of the task. We assume that when a data flow graph has multiple ingress modules then each ingress is assigned to a separate task, with packets passing between tasks over double-ended queues. Each task uses run-to-completion scheduling, and there is a separate CPU core assigned per task. Consequently, in a multicore scenario we have as many tasks as there are cores.

Flows. A flow $f = (p_f, R_f, D_f), f \in \mathcal{F}$ is an 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 task 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. 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. During this traversal, flow goes through tasks. A *taskflow* is a part of a flow that is executed on a single task.

B. System Variables

We use a fluid model. Thus, variables are continuous and differentiable, describing system *statistics* over a longer period of time that we call the *control period*. We use the following variables to describe the state of the data flow graph in a given control period (dimensions indicated in brackets). The variables needed for the multicore extension are marked by ☛.

Batch rate x_v [$1/s$]: the number of batches per second entering the network function in module v (see again Fig. 3).

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

Packet rate r_v [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

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

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 module service time profile.

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 on its task.

☛ **Task turnaround-time** τ_t [sec]: Turnaround-time of task t is the time while task t processes a packet batch. This is the multicore equivalent of the *turnaround-time* (see [10] for details). We consider the time to execute *all* task modules on maximum sized batches as an upper bound:

$$\tau_t \leq \sum_{v \in V} (T_{v,0} + T_{v,1} B) \quad \forall v \in V_t \ . \qquad (2)$$

☛ **Taskflow** π: For each flow $f \in \mathcal{F}$, π_f is a list of tasks the packets of f traverse in the data flow graph.

☛ **Per-task flow delay budget** $\Delta_{t,f}$ [sec]: delay allocated for a taskflow of flow f in task t; *i.e.,* the maximum delay allowed for a flow to traverse a task. A column vector representing delay budgets for each flow of task t is noted as Δ_t.

C. 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; see [10] for a detailed justification and several ideas to overcome them. New assumptions added for the multicore setting are marked by ☛.

Feasibility. We assume that the pipeline runs on a single task and this task has enough capacity to meet the delay SLOs.

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.

Static flow rate. All flows are considered constant-bit-rate during the control period (usually in the millisecond time frame).

☛ **Task-exclusive modules:** Each module is assigned to exactly one task. If a module needs to present in multiple tasks, it will be replicated for each task.

III. Batchy Decomposition

Decomposition is a general framework for breaking down complex optimization problems into simple *subproblems*, which are assumed to be easy to solve in separation, and a *global problem* that orchestrates the subproblems and takes care of the "complicating constraints" [14]. Each subproblem is defined in terms of a set of *private variables*, which appear only in this subproblem, and a set of *public variables* that are common to multiple subproblems. The problem is solved iteratively: first we fix the public variables and solve each subproblem separately to find the optimal setting of the private variables under the current setting of the public variables, and then in a "master step" we update the public variables and start a new iteration. The update drives the system in a direction so that the global objective is improved, *e.g.,* moving along the objective function gradient with a pre-defined step size. Depending on the type of the public variables, we distinguish *primal* decomposition and *dual* decomposition frameworks. In primal decomposition the public variables are primal variables, while in dual decomposition the subsystems are manipulating dual variables (*i.e.,* prices) of the global problem.

To demonstrate the two methods, consider an example of a printed circuit board, where the board is the global system and the integrated circuits on the board are the subsystems. Suppose we want to design a complex circuit from subcircuits (*e.g.,* integrated circuits), and our goal is to minimize the overall power usage. Subcircuits have properties, some of them are not relevant to how they connect to each other (*e.g.,* dimensions), some are important in the interconnection (*e.g.,* power usage). In this case, we say dimension is a private variable, and power usage is a public variable of the subcircuits. Then, in primal decomposition we fix the amount of power usage available to each subcircuit and design the subcircuits according to that specification. Then, we update the public variables (*i.e.,* the subcircuits' power budgets) in a way as to improve the overall power usage and then we restart the iteration, by redesigning the subcircuits (*i.e.,* solving the subproblems) subject to the new power budget. In dual decomposition, we allow subcircuits to choose how much power they want to use, however, each subcircuit has to "pay" a certain price for power usage. The price depends on the system-wide power budget: when the current power usage is low the prices are also low, but as the system's power constraints become more and more tight so do the per-unit power usage price goes up. In the global step we set the prices in a way to improve the design.

In general, if the global problem is optimized using the subgradient method then decomposition methods are guaranteed to converge "close" to the optimum, even for a constant step size [14]. With a dimisihing step size rule, arbitrary close convergence to the optimum is guaranteed in finite steps.

A. Batchy: Multicore System Model

We extend Batchy to the multicore setting by formulating the global problem for multiple cores and then applying primal decomposition to the system to obtain per-core controllers. The global problem sets the per-core delay budgets so that end-to-end delay SLOs are met and the total system load is minimized. Subproblems in turn control the batch size over a partition of the data flow graph, subject to the delay budgets set by the global problem. Then, private variables are the per-module queue sizes while the public variables are the per-task delay budgets (*i.e.,* the maximum time allowed for processing a flow in a task). In the following sections we provide further detail.

First, we recap the original Batchy model implementing single-core execution [10]. As (3) shows, we express system load L as a function of queue backlogs while conforming delay requirements (4) and queue sizing limits (5) in a single task.

$$L = \min \sum_{v \in V} \frac{R_v}{b_v}(T_{0,v} + T_{1,v} b_v) \tag{3}$$

$$\text{s.t. } \tau + \sum_{v \in P_f} (\frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v) \le D_f \qquad f \in F \tag{4}$$

$$1 \le b_v \le B \qquad v \in V \tag{5}$$

Next, we extend the single-core model to the multicore setting. For this purpose, we break up the data flow graph to tasks, under the assumptions of §II-C. The problem decomposes on a per-task basis as shown in (6)–(10).

$$L = \min \sum_{t \in \mathcal{T}} \sum_{v \in V_t} \frac{R_v}{b_v}(T_{0,v} + T_{1,v} b_v) \tag{6}$$

$$\text{s.t. } \tau_t + \sum_{v \in P_{f,t}} (\frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v) \le \Delta_{t,f} \qquad f \in F, t \in \mathcal{T} \tag{7}$$

$$\sum_{t \in \mathcal{T}} \Delta_{t,f} \le D_f \qquad f \in F \tag{8}$$

$$1 \le b_v \le B \qquad t \in \mathcal{T}, v \in V_t \tag{9}$$

$$\Delta_{t,f} \ge 0 \qquad f \in F, t \in \mathcal{T} \tag{10}$$

Next, we show the global and subproblem objectives of our decomposition. In this context, we use the term *problem* and *task* interchangeably due to the per-task decomposition.

B. Global Problem

In our primal decomposition structure the global problem is responsible for distributing the flow delay budgets among tasks in a way to minimize system load (11). We also need to ensure the sum of per-task delay budgets are not over the flow delay budget (12) and each task will receive non-negative flow delay budgets (13).

$$L = \min \sum_{t \in \mathcal{T}} L_t(\Delta_t) \tag{11}$$

$$\text{s.t. } \sum_{t \in \mathcal{T}} \Delta_{t,f} \le D_f \qquad f \in F \tag{12}$$

$$\Delta_{t,f} \ge 0 \tag{13}$$

C. Subproblems

Subproblems optimize task performance, while keeping delays under the per-task flow delay budgets assigned by the global problem. We observe that the resultant control problem is effectively the same as the single-core control problem (3)–(5). Therefore, we will mostly reuse the original Batchy controller from [10] with minimal changes to handle the private/public variables and per-core delay budgets.

We need a framework to distribute the per-flow delay budgets $\Delta_{t,f}$ across the tasks $t \in \mathcal{T}$ traversed by f. Correspondingly, for every flow there is a dedicated *leader* task that sets the per-task delay budgets, and zero or more *follower* tasks that merely track the budgets assigned by the leader. Each task may be a leader for any flow and follower for others. We categorize tasks $\forall t \in \mathcal{T}$ in the system:

- $\Omega_t = \{f : t \text{ is the leader for } f\}$,
- $\Psi_t = \{f : t \text{ is a follower for } f\}$.

The fundamental difference between leaders and followers is that a leader keeps track of the per-task flow delay budget subgradients along the flow path: $\Theta_{t,f} : f \in \Omega_t, s \in \mathcal{T} : f \in \Psi_s$. Leaders use both subgradients and queue size backlogs $b_v : v \in V_t$ as private variables. Likewise, followers use queue backlog sizes as private variables, and per-task flow budgets $\Delta_{t,f}$ as public variables.

Take the pipeline of Fig. 2 as an example; we have a single flow f_1 passing over 3 tasks $\mathcal{T} = t_1, t_2, t_3$. Select t_3 as the leader of f_1, so $\Omega_{t_3} = \{f\}$. Consequently, t_1 and t_2 will be followers of f_1. Leader private variables are the delay budget subgradients Θ_{t_1,f_1} and Θ_{t_2,f_1}, and the queue backlog sizes $b_v, v \in V_{t_3}$. Followers tasks optimize their private variables b_v according to public variables: Δ_{t_1,f_1} or Δ_{t_2,f_1}.

The subproblem objective function (14) minimizes task load; in this manner it is equivalent to the single-core objective function. Private delay budget variables $\Theta_{t,f}$ are not effecting the task load, therefore are omitted from the objective function.

$$L_t(\Delta_{t,f}) = \min l_t = \min \sum_{v \in V_t} \frac{R_v}{b_v}(T_{0,v} + T_{1,v} b_v) \qquad (14)$$

The objective function is subject to the following constraints. For both leader and follower problems, the batch size limiting constraint (15) applies.

$$1 \le b_v \le B \qquad v \in V \qquad (15)$$

Additionally, flows passing the task must meet their delay SLO requirement. The constraints are slightly different for leader and follower problems. As of follower problems, constraint (16) keeps per-task flow delays under the budget ($\Delta_{t,f}$). Recall, these budgets come from the global problem (11).

$$\tau_t + \sum_{v \in P_{f,t}} \left(\frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v\right) \le \Delta_{t,f} \qquad f \in \Psi_t \qquad (16)$$

Leader problems have multiple delay constraints. First, constraint (17) ensures compliance with delay SLOs of both taskflows and flows. This is doable since leader tasks have a view on private delay variables ($\Theta_{t,f}$). Second, constraint (18) ensures equivalence between public and private delay variables.

$$\tau_t + \sum_{v \in P_{f,t}} \left(\frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v\right) + \sum_{s \in \mathcal{T}: f \in \Psi_s} \Theta_{t,f} \le D_f \qquad f \in \Omega_t \qquad (17)$$

$$\Theta_{t,f} = \Delta_{t,f} \qquad f \in \Omega_t, s \in \mathcal{T} : f \in \Psi_s \qquad (18)$$

IV. Control Algorithms

In this section we present efficient control algorithms to solve both the global problem and the subproblems. These algorithms are suitable for a real-life implementation.

A. Solving Subproblems

Batchy uses a controller based on the gradient projection method of Rosen [21]. The Rosen method is compatible with our decomposition: it handles equality-type constraints (18) and generates gradients and dual variables for the subgradient method, which are used in the subgradient step for solving the global problem (see later in §IV-B).

Let us briefly recap the gradient projection method. The method consists of three main steps: *i)* find an improving direction; *ii)* find a suitable step size; *iii)* optimize along the direction with the step size. In the first step, we obtain an improving feasible direction by projecting the gradient of the objective function into the feasible space using a projection matrix. The projection matrix **P** ensures that the resultant update will not violate the per-task delay budgets. For this end, we show the construction of variable coefficients matrix **M** and the projection matrix **P**:

- Let $\mathbf{M_1} = [AB]$ be a matrix where A is a matrix in which row i reflects the effect of increasing queue backlog sizes (b_v) on i-th flow delay in $\mathcal{F}$ with tight delay constraints from (16) and (17), and B is a zero matrix corresponding to the private variables $\Theta_{t,f} : f \in \Omega_t, s \in \mathcal{T} : f \in \Psi_s$.
- Let $\mathbf{M_2} = [ZQ]$ where Z is a zero matrix with as many rows as there are constraints in (18) and as many columns as the number of task modules $|V_t|$ (corresponding to b_v variables). Q is a matrix with row i set to 1 where $\Delta_{t,f}$ is the i-th taskflow in a list of taskflows $t, f : f \in \mathcal{F}, t \in \mathcal{T}$.
- Let $M^T = [M_1^T M_2^T]$.
- Then, we construct **P** as $\mathbf{P} = I - M^T(MM^T)^{-1}M$.

The subproblem control algorithm reuses the single-core Batchy control algorithm with the new projection matrix **P**. The control algorithm generates the duals of private delay variables $\Theta_{t,f}$ (ω) for the sugradient method by the leader task of flow f. We summarize the per-task projected gradient control algorithm we use to solve the subproblems in Algorithm 1.

Unfortunately, the control algorithm cannot handle an infeasible state; *i.e.,* a state where the SLOs cannot be met. To recover the system from infeasibility we introduce a simple heuristic: the subsystems reuse the feasibility-recovery mechanisms from single-core Batchy [10], while multicore feasibility-recovery is implemented in the global controller (§IV-B).

Algorithm 1 Projected Gradient Control Algorithm

```
procedure ProjectedGradient($\mathcal{G}, \mathcal{F}, \Delta_t, f$)
    ▷ Gradient projection
    while $True$ do
        $\mathbf{P} = \mathbf{I} - \mathbf{M}^T(\mathbf{M}\mathbf{M}^T)^{-1}\mathbf{M}$
        $\Delta\mathbf{b} = \mathbf{P}\nabla l_t$   ▷ $\Delta\mathbf{b}$ has useless coordinates corresponding to private variables $\Theta_{t,f}$
        $\mathbf{w} = -(\mathbf{M}\mathbf{M}^T)^{-1}\mathbf{M}\nabla\mathbf{l} = [u, \omega]$   ▷ $u$ corresponds to $b_v$ and $\omega$ corresponds to $\Theta_{t,f}$
        if $\Delta\mathbf{b} \ne 0$ then break
        if $\mathbf{u} \ge 0$ then return   ▷ Optimal KKT point reached
        delete row for $f$ from $\mathbf{M}$ for some $f \in \mathcal{F} : w_f < 0$
    ▷ Line search
    for $v \in V, f \in p_v$ do
        if $\Delta b_v > 0$ then
            $\lambda_v = \min_{f \in \mathcal{F}: v \in p_f} \left\lceil \frac{\Delta_{t,f} - \tilde{t}_f}{\Delta b_v} \right\rceil$
    $\lambda = \min_{v \in V} \lambda_v$
    for $v \in V$ do SetTrigger($v$, $b_v + \Delta b_v \lambda$)
```

B. Solving The Global Problem

Subproblems are handled by the Batchy projected gradient controller at each control period. After every N iteration, the global problem controller kicks in to reallocate the per-task delay budgets (*i.e.,* the public variables $\Delta_{t,f}$).

The global control algorithm relies on two types of inputs: the duals ω_n of the subproblem constraints (16), and duals ω_m of constraints corresponding to private variables in (18). Gradients g are obtained by summing global and subproblem subgradients pairwise: $g_{n,f} = \omega_m + \omega_n \ \forall f \in \Omega_m, n \in \mathcal{T} : f \in \Psi_n$. Based on these inputs, the global control algorithm (Algorithm 2) first calculates a step size, then updates per-task delay budgets for each flow. For simplicity, the algorithm uses a fix step size calculated as a configurable percentage δ of flow delay D_f.

We apply a simple heuristics to prevent infeasible states in the global problem. We collect taskflows that exceed their delay budget and increase their budget with a configurable and fixed percentage of flow delay, balancing this delay increment

Algorithm 2 Global Control Algorithm

```
procedure SUBGRADIENT GLOBAL STEP($\mathcal{G}, \mathcal{F}, \pi, \delta, g$)
    for $f \in \mathcal{F}$ do
        ▹ Calculate step size
        $\alpha = D_f * \delta$
        ▹ Update allocated per-task flow delays on $\pi_f$
        for $t \in \pi_f$ do
            $\Delta_{t,f} = \Delta_{t,f} + \alpha * g_{t,f}$
```

Table I
STEADY-STATE RESULTS (SIMPLE PIPELINE).

	Rate [Mpps]	**Delay (p99)** [μs]
No Batching	0.971	15.445
Static Delay Budgets	0.991	18.615
Multicore Batchy	*1.348*	*11.255*

by decreasing surplus budgets of the feasible taskflows. This simple technique is sufficient to ensure flow delay SLOs (12) and non-negative per-task flow delay budgets (13).

V. EVALUATION

In this section, we evaluate our Batchy multicore extension on both synthetic example and real-life use-case. We reused existing Batchy codebase [10] as a controller to solve the per-task subproblems (Algorithm 1) and implemented the subgradient controller to orchestrate the per-task Batchy controllers (Algorithm 2). The source code is available on GitHub [18]. The evaluation was running on a server with 6×2.4GHz CPU (power-saving disabled) and 64GB RAM installed with Debian 11 GNU/Linux.

A. Concept Validation: A Simple Pipeline

The first evaluation scenario focuses on validating the concept.

Evaluation setup. We use a simple pipeline of two tasks connected back-to-back. Tasks run on different cores and contain one module. The system has one flow that traverses both tasks. The last module is a computation-heavy module that requires significantly more per-batch processing time (tens of thousands of CPU cycles) than the first module (hundreds of CPU cycles). This pipeline is similar to the example in Fig. 2.

We compare multicore Batchy to two baselines. The first baseline does no packet batching. The second baseline runs Batchy, but does not adjust per-task delay budgets adaptively; *i.e.,* adopts the naïve approach of Fig. 2. The measurements focus on steady state performance: the first 100 control periods are considered as warmup time, and we focus on the next 100 control periods. The flow delay SLO is set to $12\mu s$.

Results. Table V-A summarizes steady packet rate and 99th percentile delays of the measurements. The two baselines produce limited packet rate due to poor batch-scheduling algorithms. Namely, baselines cannot mitigate the cost of computation-heavy task by intensive batching. There is a slight difference between the performance of the two baselines: in case of *static delay budgets*, Batchy has enough room for batching in the first task, yielding a slight overall improvement of the packet rate at a 20% delay penalty. In contrast to baselines, *multicore Batchy* can distribute the global delay bound across the tasks optimally, so that it assigns extra delay budget surplus for the last task that enables it to execute the computation-heavy module on larger batches. This optimization improves throughput by 30% while decreases delay by 60%, and makes multicore Batchy the only solution to meet the flow delay SLO.

To sum up, this experiment highlights the importance of batching in multicore scenarios. However, careful distribution of delay budgets among processing cores is necessary to get the most of batch-efficiency gains.

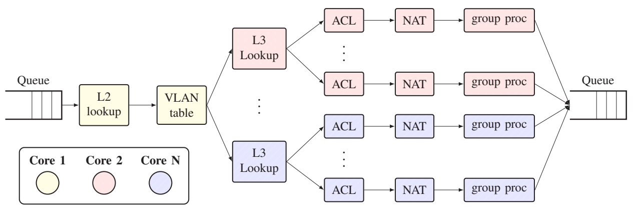

Figure 4. The *Virtual Routing Function* Pipeline on N Cores.

B. Case Study: Virtual Routing Function

We demonstrate the real-life applicability of multicore Batchy on a sample use case, the Virtual Routing Function (VRF), taken from an official 5G benchmarking suite [13]. In this measurement we are focusing on the following questions: *i)* can we decrease the delay compared to single-core Batchy; *ii)* how efficient is the decomposition-based delay budget distribution compared to a naïve approach; *iii)* how much extra processing is required for the hierarchical control?

Evaluation setup. The VRF pipeline (Fig. 4) implements a latency-optimized L2/L3 routing scenario often arising in the context of network function virtualization. In addition to L2/L3 routing, the pipeline also performs access control and address translation over multiple virtual LANs (VLANs). First, traffic is split per VLANs, and then for each VLAN the next hop is selected using longest-prefix matching (L3 Lookup). For each next hop, traffic undergoes access control (ACL), address translation (NAT), and group processing. The pipeline has two parameters: the number of VLANs (n), and the number of next-hops per VLAN (m). The pipeline is provisioned on $n+1$ cores: VLAN splitting is done on the first core, and per-VLAN traffic is processed on the remaining n cores.

For the evaluation, we use the VRF(2,4) pipeline (2 VLANs and 4 next-hops/VLAN). We set a $72\mu s$ delay SLO for all flows. The system runs for 60 control periods, and each control period takes 0.5s. The global controller kicks in at every 10th period; this gives enough time to the per-core (subproblem) controllers to adapt to new delay budgets. We compare single-core Batchy, naïve multicore (static per-task delay budgets), and full-fledged multicore Batchy.

Results. Fig. 6 shows the key performance indicators (*i.e.,* rate and delay) in the system. Naïve and Batchy multicore approaches start from the same initial state. Yet, the full-fledged multicore Batchy is able to further improve the performance by adjusting the per-task delay budgets. Fig. 5 shows the underlying control loops: *i)* the global controller takes the surplus delay budget of the VLAN splitting task and gives extra

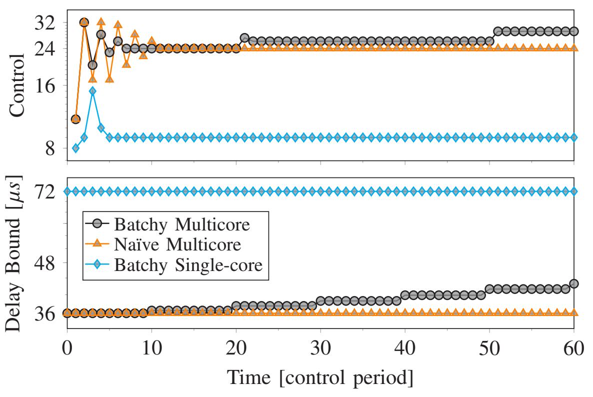

Figure 5. Control Parameters of the First Flow in VRF(2,4): control for ACL module and per-task delay budget on the second core (recall Fig. 4).

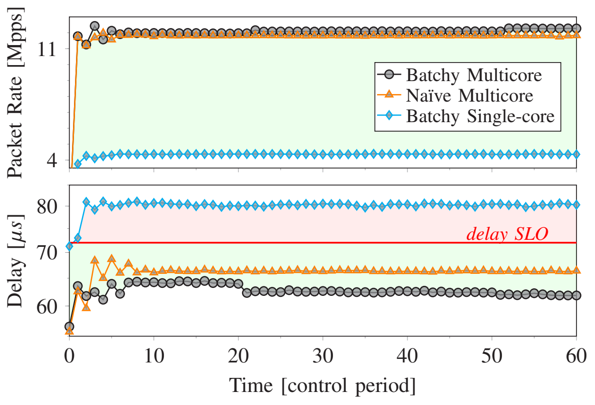

Figure 6. Key Performance Indicators of the VRF(2,4) pipeline: total packet rate and delay of the first flow. Delay SLOs are set to 72μs.

budget to the processing-heavy per-VLAN traffic processing; *ii)* this gives enough time to the per-VLAN task controller to queue up larger packet batches. This coordinated optimization improves the overall performance (Fig. 6). Over single-core Batchy, packet rate increases 2.5× and flow delay reduces to 0.75×. More importantly, the delay is finally below the SLO!

As of the controller performance, we measured the per-task and global controllers running time in each control period during the measurement and found that the multicore approaches result only a 7% increase on average due to extra global control steps.

To conclude, we see that our multicore extension is an enabler technology for Batchy, supporting use cases with ultra-low delay SLOs. We see the decomposition improves performance and its control overhead is negligible.

VI. Related Work

A. Optimizing Resource Usage

Carefully execute an NF-chain on general-purpose hardware is one way to achieve performance improvement. Shenango [22] improves CPU utilization by bypassing the kernel and reschedule or scale up according to the occupancy of the packet ring buffers. This technique results low latency and improved CPU utilization Similarly, IX [6] utilizes adaptive batch control to improve throughput and latency. Metron [20] improves end-to-end performance in NF-chains by avoiding cross-CPU issues in NF-scheduling. These works focus on optimizing performance without controlling latency. In contrast, Batchy not just improves performance but carefully controls latency to meet SLOs.

B. Improving Performance by Offloading

Offloading some part of the processing to hardware components such as SmartNICs [23], FPGAs [24], or GPUs [8], [25], [26] is widely used to improve packet processing performance. To mitigate the packet offloading cost and to maximize GPU utilization, extensive batching [26] and careful load balancing between the offload hardware and the CPU [25] are required. Offloading works motivate the importance of batching, however, they are orthogonal to our work since they incorporate offloading to specific hardware elements.

C. Meeting Delay SLOs

Beside performance optimization, guaranteeing SLOs is another highly-desired behavior of NFV systems. Grus [8] an NFV framework with GPU offload introduces a multi-layer system with admission control and latency prediction model to guarantee delay SLOs. As opposed to our work, Grus guarantees delay SLO only for single VNF deployments, and the model is tailored for the GPU offloading scenario. SLOMO [27] predicts potential performance of VNF colocation, but does not provide SLO guarantees. In contrast to Grus, ResQ [28] provides performance isolation at CPU last-level cache solving the noisy neighbor problem of VNFs, and enables enforcing SLOs. NFV-RT [29] provides soft real-time guarantees for NF service chains deployed in data center environment using a fat-tree topology.

As opposed to our controller framework running on general hardware, these works are bound to a given NFV environment: they require a certain CPU feature, or specific underlying network topology. Our work focuses on a single host using general CPUs. Moreover, our controller framework 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, and selective SLO-enforcement at the granularity of individual flows in multicore systems.

VII. Conclusions

Batchy, a state-of-the-art batch-scheduling framework, presents massive performance improvements while conforming delay SLOs even at Mpps-scale traffic with SLOs at μs range. Batchy focuses on single-core execution.

In this paper we introduce a multicore extension to Batchy. To this end, we formulated a primal decomposition to find the optimal run-to-completion batch-scheduling on multicore systems. We developed and implemented effective control algorithms to be used in practical data flow graph batch-scheduling. Our evaluation on a real 5G use-case focusing on latency-optimized network function virtualization shows that the multicore Batchy provides better performance (2.5×

packet rate) over the single-core algorithm and guarantees delay SLOs that are otherwise not feasible with the single-core algorithm.

Future work focuses on applying Batchy for ultra-low-latency and real-time applications in 5G and beyond networks.

ACKNOWLEDGMENT

This work was supported by the ÚNKP-21-4 New National Excellence Program of the Ministry of Innovation and Technology from the source of the National Research, Development and Innovation Fund, and NKFIH/OTKA Project #135606.

REFERENCES

[1] 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*, vol. 107, no. 4, pp. 746–764, 2019.

[2] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[3] E. Warnicke, "Vector packet processing - one terabit router," July 2017.

[4] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE ANCS*, 2015, pp. 5–16.

[5] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane specialization for high-performance OpenFlow software switching," in A*CM SIGCOMM*, 2016, pp. 539–552.

[6] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 4, p. 11, 2017.

[7] Intel, "Data plane development kit," http://dpdk.org.

[8] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, and J. Wu, "Grus: Enabling latency SLOs for GPU-accelerated NFV systems," in *IEEE ICNP*, 2018, pp. 154–164.

[9] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," in *ACM SIGCOMM*, 2017, pp. 71–84.

[10] T. Lévai, F. Németh, B. Raghavan, and G. Rétvári, "Batchy: Batch-scheduling data flow graphs with service-level objectives," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 633–649. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/levai

[11] J. Nagle, "Congestion control in IP/TCP internetworks," Internet Requests for Comments, RFC Editor, RFC 896, January 1984.

[12] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv*., vol. 54, no. 4, May 2021. [Online]. Available: DOI: 10.1145/3447868

[13] 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*, vol. 36, no. 12, pp. 2621–2630, Dec. 2018.

[14] S. Boyd, L. Xiao, A. Mutapcic, and J. Mattingley, "Notes on decomposition methods," *Notes for EE364B, Stanford University*, vol. 635, pp. 1–36, 2007.

[15] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1439–1451, 2006.

[16] D. Palomar, "Convex primal decomposition for multicarrier linear mimo transceivers," *IEEE Transactions on Signal Processing*, vol. 53, no. 12, pp. 4661–4674, 2005.

[17] H. Pennanen, A. Tolli, and M. Latva-Aho, "Decentralized coordinated downlink beamforming via primal decomposition," *IEEE Signal Processing Letters,* vol. 18, no. 11, pp. 647–650, 2011.

[18] "Batchy," https://github.com/hsnlab/batchy/tree/multicore.

[19] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *USENIX NSDI,* Feb. 2019, pp. 33–46.

[20] 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," in *USENIX NSDI*, 2018, pp. 171–186.

[21] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear programming: Theory and algorithms*. John Wiley & Sons, 2013.

[22] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX NSDI*, 2019, pp. 361–378.

[23] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman, "Uno: Uniflying host and smart nic offload for flexible packet processing," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, p. 506–519.

[24] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conferenc*e, ser. SIGCOMM '16. ACM, 2016, p. 1–14.

[25] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "NBA (Network Balancing Act): A high-performance packet processing framework for heterogeneous processors," in *EuroSys*, 2015, pp. 22:1–22:14.

[26] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A GPU-accelerated software router," in *ACM SIGCOMM*, 2010, pp. 195–206.

[27] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *SIGCOMM '20*, 2020, p. 270–282.

[28] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in network function virtualization," in U*SENIX NSDI*, 2018, pp. 283–297.

[29] Y. Li, L. T. Xuan Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *IEEE INFOCOM 2016*, 2016, pp. 1–9.

Tamás Lévai received the M.Sc. degree in Computer Engineering at the Budapest University of Technology and Economics (BME) in 2016. Currently, he is a Ph.D. candidate and assistant teacher at BME. His research interest focuses on computer networks and distributed computing, mainly software-defined networking, cloud native computing and high-performance packet processing.

Gábor Rétvári received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology and Economics in 1999 and 2007. He is now a Senior Research Fellow at the Department of Telecommunications and Media Informatics. His research interests include all aspects of network routing and switching, the programmable data plane, and the networking applications of computational geometry and information theory. He maintains several open source scientific tools written in Perl, C, and Haskell.