EMBEDDING AND BATCH-SCHEDULING
DATA FLOW GRAPHS IN SOFTWARE SWITCHES

Tamás Lévai

Summary of the Ph.D. Dissertation

Supervised by

Gábor Rétvári, D.Sc.
Senior Research Fellow

Budapest, Hungary
2022
1 Introduction

*Programmable software switches* are fundamental building blocks of the modern info communication network, deployed in a wide range of applications from (virtual) data centers to telecommunication networks. Current constantly evolving network applications cannot be implemented by legacy fixed-function network components. Programmable network components, on the contrary, are able to be rapidly reconfigured in concert with current demands of constantly evolving applications and protocols. Consequently, instead of following the legacy approach, modern telco use cases, industry use cases, clouds, and beyond rely on Software-Defined Networking (SDN) and Network Function Virtualization (NFV) due to their flexibility and moderate costs. In these networks, the network functions are realized in software on general-purpose or re-programmable hardware. Application logic is described as a packet processing pipeline of interconnected (virtual) network functions. The network architecture consists of separate control plane and data plane. The control plane manages the pipeline; the data plane executes the pipeline. Application changes are handled instantly by re-configuring the packet processing pipeline.

The pipeline is usually abstracted as a *data flow graph*: a directed graph in which the nodes are network functions and arcs describe control flow. *Data-flow-graph embedding* is responsible for assigning packet processing primitives (i.e., network functions, modules) to workers (e.g., CPU cores). Meanwhile, *data-flow-graph batch-scheduling* selects the next packet processing primitive to be executed on a packet batch.

Modern use cases require high-throughput, low-latency, and high-availability from software switches. These performance requirements are described by Service Level Objectives (SLOs). SLOs consist of a service-level indicator (e.g., one-way end-to-end latency) and a measurement methodology (e.g., 99th percentile values of periodic measurements). For example, an industry automation use case requires one-way, end-to-end, 99th percentile latency under 10msec.

High-availability applications such as robot motion control or voice on a mobile gateway require the software switch to operate securely even when a CPU failure or software error occurs. In the data-flow-graph context, these events risk the connectivity of the graph, which results in service outages and violation of the availability-SLO.

Unfortunately, current data-flow-graph embedding and batch-scheduling methods were not conceived with these requirements in mind. In this dissertation, we are improving and extending software switch performance and improve data-plane availability using novel approaches to optimizing data-flow graph embedding and batch-scheduling.

Processing packets in batches is the essence of data flow graph batch-scheduling. Figure 1 shows a motivating example for our work. Figure 1/a shows an example in which the data flow graph receives full packet batches. The first network function (A) processes packets in a single step, then pushes them to downstream network functions (B in this case). However, batches tend to fragment if a function has multiple downstream functions (see Figure 1/b). Consequently, the efficiency of downstream functions decrease since they operate below full-capacity. We can mitigate this issue by defragmenting batches inside the data flow graph. As Figure 1/c shows, this method involves buffering. Buffering introduces additional delay which might violate delay SLOs.

In the first part of the dissertation, we present a comprehensive mathematical model to describe data-flow-graph batch-scheduling. We use this model to formulate optimal batch-scheduling.
with delay-type Service Level Objectives (SLOs) on the flows traversing the switch. We introduce a batch-scheduling controller framework to reach optimal batch-scheduling with delay-SLOs on a real-life software switch under real workloads. We show our controller attains $2 - 3 \times$ improvement while conforming to strict delay-SLOs on the $\mu$sec-scale in modern use cases and beyond.

Optimal batch-scheduling requires a proper data-flow-graph embedding. The task of data-flow-graph embedding in this context is concerned with assigning each packet processing module of the data flow graph to a worker, i.e., CPU core, in a way as to guarantee 3 crucial objectives: feasibility, efficiency, and resiliency. A feasible embedding prevents over assigning CPU cores, in order to preclude CPU overloads that may result in significant service degradation. An efficient data-flow-graph embedding is one that minimizes the number of CPU–CPU crosslinks along service chains to prevent the the huge toll on inter-CPU-core links [9, 10] (common performance bottleneck of multi-core systems). A resilient data-flow-graph embedding method will ensure that the critical packet processing paths in the software switch are immune to CPU failures. The key idea is to separate the dataflow graph to high-availability modules, which have the property that the packet processing path of at least one high-availability flow/service chain uses the module, and the rest of the modules that serve bulk flows only, and then provide resilience only for these high-availability modules by duplicating the corresponding paths in the dataflow graph. This way we can save remarkable amount of resources as compared to naïvely duplicating the entire dataflow graph. A motivating example for resilient embedding is shown in Figure 2.

In the second part of the dissertation, we consider reliable data-flow-graph embedding. In this part, the main challenge is to allocate CPU resources to each module in the data flow graph so that packet processing is immune to hard and soft CPU errors, which would otherwise break data-plane connectivity and cause service outages. We formulate an exact algorithm for optimal embedding and show fast and effective heuristics to solve the problem in polynomial time.
2 Research Goals

Modern info-communication applications call for strict requirements on the network. In order to understand how off-the-shelf state-of-the-art software switch frameworks are suitable to answer these challenges, we conducted a comprehensive multi-year evaluation of the available software switch products. [J2]. Our main observation was that current software switches lag behind fixed function switches and middleboxes, concerning performance characteristics and critical features.

Our research aims to improve the performance characteristics and features of software switches. We focus on the main component of programmable software switches: their packet processing pipeline expressed as data flow graph. The embedding and scheduling of this data flow graph determines the performance and resilience of the software switch. To improve both embedding and scheduling of data flow graphs in software switches, our research focuses on the following points:

- Study data-flow-graph batch-scheduling in software switches. Analyze and model packet processing execution, identify performance limiting factors such as batch fragmentation, and highlight the lack of delay-SLO guarantees. Formulate data-flow-graph batch-scheduling that can de-fragment batches and thus improve performance and guarantee delay-SLOs.

- Adapt the optimal schedule to real use cases on real software switches. Handle the dynamic nature of modern use cases by an online solution: a controller. Design a batch-scheduling controller and integrate it with a real software switch.

- Study data-flow-graph embedding in software switches. Assign main embedding objectives to fulfill requirements of modern use cases. Formulate embedding that can guarantee high-performance and resilience against CPU errors. Study naïve algorithms for fast and efficient embedding.

3 Methodology

Analytical methods were applied to describe batch-scheduling and embedding of data flow graphs. In our model (Thesis 1), a directed acyclic graph represents the data flow graph. We expect a homogeneous architecture of packet processing CPU cores, hence we describe cores with their quantity and capacity. The model requires profiling data to predict performance and test SLO conformance. The profiling data is gathered by profiling packet processing modules individually offline and monitoring the software switch online. We validate our results on multiple software switches, network functions, and numerous data flow graphs of real-life applications.

We designed a convex program to describe the optimal data flow graph batch-scheduling. The convex program is solved in real software switches by a single-step model-based controller that can adapt to the dynamic nature of modern applications.

Data-flow-graph embeddings were formulated as decision problems. We show they are NP-hard, and we give an ILP formulation. We present a family of fast and effective heuristics for finding feasible and efficient embedding, and resilient embedding in polynomial time, and prove their efficiency by extensive evaluation including a numerical evaluation and a case study on a real software switch.
Performance analysis of data flow graphs on reference data flow graphs of real 5G use case on multiple software switches were conducted with TIPSY [C1], a complex benchmarking tool we developed.

The data flow graph batch-scheduling controller framework of Thesis 2 was implemented in Python, and integrated with BESS [11] software switch. We developed new BESS modules (e.g., FractionalBuffer, and BatchMeasure). For the evaluation, Python and BASH scripts were used.

The data flow graph embedding algorithms presented in Thesis 3 were implemented in C++ using the LEMON [12] graph library and Gurobi [13] solver. Case study was conducted on BESS.

Software components and artifacts are available under GPL free software license in order to facilitate reusing and reproducing our results [J1, C2, C3].

3.1 Frequently Used Data Flow Graphs

The following 5G use case data flow graphs were used for batch-scheduling and embedding evaluations.

**Industry floor automation/Robot control (RC).** A 5G application combining Industry 4.0 tasks (e.g., Augmented Reality, and robot motion control) on a distributed system. These tasks present strict requirements on the network. For example, robot control traffic requires ultra-low latency on milliseconds scale, therefore are usually implemented on premise as an edge-computing platform on which traffic tends to mix at software switches making SLO guarantees hard to achieve.

To model this application, we introduce an a 5G robot control pipeline RC($n$) (see Figure 3) with $n$ branches, with the upper branch representing an ultra-delay-sensitive industry automation service chain with ultra-low delay-SLO and the rest of the branches carrying bulk traffic.

![Figure 3: Robot Control Pipeline.](image)

**L2/L3 (Virtual) Routing Function (L2/L3, GW, VRF).** Routing or Virtual Routing on L2/L3-level is widely-used application. To model various L2/L3 (virtual) routing use cases, we introduce the following data flow graphs (see Figure 4): 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.

**Mobile GateWay (MGW).** The Mobile GateWay is a key component in a 4G/5G telecommunications operator infrastructure. It connects a set of mobile user equipment (UEs), located behind
base stations (BSTs), to a set of public servers available on the Internet. The pipeline is asymmetric due to packet (de)capsulation. In the uplink direction (UE/BST -> server) the MGW receives GTP-encapsulated packets from the base stations, identified by the source IP address in the GTP header, and forwards the decapsulated packets to public servers in the Internet. In the downlink direction (server -> UE/BST) the MGW receives normal packets from the Internet and the pipeline is basically the reverse of the uplink one. On both directions, multiple bearers are defined. These bearers carry different traffic classes (e.g., voice call or data transfer). Consequently, the bearers have different service level requirements. Additionally, the gateway implements accounting and logging too.

Our MGW\((m,n)\) pipeline represents a full 5G mobile gateway data plane with \(m\) users and \(n\) bearers per user, with complete uplink and downlink service chains (see Figure 5). We emulate uplink and downlink processing (e.g., rate limitation and accounting) with dummy modules. The first bearer is designated to process low-latency traffic, thus we define delay-SLOs on the first bearer flows (“QoS” nodes on Figure 5); other bearers process bulk traffic.
4 New Results

4.1 Optimal Batch-Scheduling with Service Level Objectives

Thesis 1 introduce an expressive mathematical model for SLO-based batch-scheduling, so that we can reason about the performance and delay in a batch-processing pipeline analytically and fine tune batch de-fragmentation subject to delay-SLOs. We also fix the set of basic assumptions under which the resultant mathematical program is convex and, consequently, the optimal schedule is well-defined. To the best of our knowledge, this is the first analytical model of run-to-completion batch-scheduling.

Thesis 1. [C2] I have designed a comprehensive analytical model for batch-scheduling in packet processing pipelines under delay constraints. The model covers run-to-completion and WFQ (weighted-fair-queuing) scheduling mode. I formulated the problem in a novel analytical model and I gave an algorithm to the optimal batch-schedule for WFQ and run-to-completion mode. Furthermore, I have shown a deep equivalence between run-to-completion and WFQ scheduling under assumptions A1, A2, A3 and A4.

4.1.1 Data Flow Graph Batch-scheduling Model

It is crucial for the model to use simple but realistic metrics for describing data flow graphs. First step, is characterizing performance of network functions/modules. Packet processing has two main cost component. First is the per-batch cost component consisting of CPU interrupts, function calls, I/O and memory management costs of processing a new packet batch. The other cost component involves processing the packets. Batching amortizes the fix cost component over multiple packets, enables compiler optimizations (e.g., loop unrolling, vector instructions) and optimizes CPU-cache usage. However, network functions benefit from batch-scheduling to varying degree due to implementation differences.

Thesis 1.1. [C2] By extensive measurements on real-life packet processing engines, I have shown that batch processing time of common-case primitive packet processing functions exhibit a linear dependence on the number of packets in each batch.

The batchness metric builds on the linearity presented in Thesis 1.1, and shows the batch-sensitivity of a network function. Let $T_{v,0}$ be the per-batch cost component, and let $T_{v,1}$ be the per-cost component for all $m \in V$. The batchiness value of a module $\beta_v \in [0, 1]$ can be calculated (assuming the batch size $B \in \mathbb{N}^+$) using (1).

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

Batchiness varies between 0 and 1; small $\beta_v$ indicates substantial processing gain on module $v$, while modules with $\beta_v \sim 1$ are dominated by the per-packet cost, thus can not benefit from processing large batches. Figure 6 shows profile measuring results and the resulting service time profile of 4 BESS modules. The coefficient of determination $R^2$ of this model is above 96% in our tests, indicating a good fit for the linear model. The relatively small batchiness measures suggest that most real-world packet-processing functions are particularly sensitive to batch size.
Figure 6: 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 DPDK-based ACL module at batch size 4, 8, and 16.

![Figure 6: Service-time Profile](image)

**Figure 6:** 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 DPDK-based ACL module at batch size 4, 8, and 16.

**Concepts**

**Thesis 1.2.** [C2] I have designed a system model to describe batch processing packet processing pipelines. I have modeled packet processing primitives as a combination of an ingress queue and a network function at the egress connected back-to-back. The model covers all types of packet processing primitives that do buffering, processing or splitting.

**Data flow graph.** We model the pipeline as a directed graph $G = (V, E)$, 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 Figure 7). **Input gates** (or ingates) are represented as in-arcs $(u, v) \in E : u \in V$ and **output gates** (or outgates) as out-arcs $(v, u) \in E : u \in V$. At execution, the network function creates a new batch from the packets it finds in the ingress queue, executes the requested operation on all packets of the batch, and then forms new sub-batches from the resultant packets and places these to the appropriate output gates. A batch sent to an outgate $(v, u)$ of $v$ will appear at the corresponding ingate, and hence in the ingress queue, 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 standard 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. If available, batching significantly improves performance. We denote the maximum batch size by $B$, a system-wide parameter. For Intel DPDK and BESS $B = 32$.

**Splitters/mergers.** Any module may have multiple ingates (merger) and/or multiple outgates (splitter), or may have no ingate or outgate at all. A packet parser module may feed multiple
differently protocol modules based on the header fields parsed out from each packet, while an IP Lookup module would distribute packets to several downstream branches, each performing group processing for a different next-hop, by matching the IP destination address against a routing table stored in the module (splitter); a NAT module may multiplex traffic from multiple ingates (merger). Certain modules are represented without ingates, such as a NIC receive queue or an in-band packet generator; we call these ingress modules. Similarly, a module with no outgates (e.g., a NIC transmit queue) is an egress module, or a sink. Denote the set of egress modules by \( O \).

**Compute resources.** A worker is an abstraction for a CPU core, where each worker \( w \in W \) is modeled as a connected subgraph \( G_w = (V_w, E_w) \) of \( G \) 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. 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.

**Flows.** A flow \( f = (p_f, R_f, D_f) \), \( f \in F \) is our abstraction for a service chain, where \( p_f \) is a path through \( G \) 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. 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 \( f \).

**System Variables**

Using the presented concepts, we define the following system variables to describe the dynamics of a running packet processing engine.

- **Batch rate** \( x_v \) [1/sec]: the number of batches per second entering the network function in module \( v \) (see again Figure 7).

- **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** \( r_v \) [pkt/sec]: 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
  \[
  t_v = t_v,\text{queue} + t_v,\text{svc} = 1/x_v + (T_v,0 + T_v,1 b_v) ,
  \]
  where \( t_v,\text{queue} = 1/x_v \) is the queuing delay by Little’s law [14] and \( t_v,\text{svc} = T_v,0 + T_v,1 b_v \) is the processing time needed to execute the network function in module \( v \) on all packets in a batch of size \( b_v \) according the module’s profile. In fact, this is the delay experienced by the last packet of the batch.

- **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 on the turnaround time:

\[ T_0 \leq \sum_{v \in V} (T_{v,0} + T_{v,1} B) \quad (3) \]

Note that the use of Little’s law to model queue latency is correct only in the case if the entire content of the queue is consumed in one step by the subsequent module. Correspondingly, we will need to ensure that any module is scheduled frequently enough so that no more than \( B \) packets line up in the ingress queue.

**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.

A1 **Lossless Execution**: We assume that the total packet rate at the pipeline ingresses equals the total packet rate at the egresses: \( \sum_{s \in S} R_s = \sum_{d \in O} R_d \). This assumption will generally hold true as long as modules never intentionally drop packets and the A3 Feasibility assumption holds, and therefore internal queues never overflow.

A2 **Single-core/Single-queue**: Single ingress on each worker. 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.

A3 **Feasibility**: We assume that the pipeline runs on a single worker and this worker has enough capacity to meet the delay-SLOs. Later, we show a heuristic method to decompose a data flow graph to multiple workers to address SLO violations stemming from inadequate resources.

A4 **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.

### 4.1.2 Optimal Batch-scheduling

In this section we formulate the optimal data-flow-graph batch-scheduling in the presented data flow graph scheduling model for weighted-fair-queueing and run-to-completion modes.

**Weighted-fair-queuing (WFQ)**

Workers typically run an explicit scheduler (i.e., WFQ scheduler) to distribute CPU time across the modules in the worker graph. [15] The WFQ scheduler runs side-by-side with the pipeline and decides which module to execute next. For simplicity, we consider an idealized WFQ scheduler, where execution order is defined in terms per-module rates and not weights.

Consequently, 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 to satisfy the SLOs. If multiple such rate allocations exist, then we aim to choose the one that
minimizes the overall system load ($l_v = x_v l_{v,svc} = x_v(T_{v,0} + T_{v,1} b_v)$). 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:

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

(4)

Once scheduled, module $v$ will process at most $b_v \in [1, B]$ packets through the network function, contributing $l_{v,svc} = T_{v,0} + T_{v,1} b_v$ delay to the total latency of each flow traversing it. 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 a single worker and each queue holds at most $B$ packets), and also using (2), we get the following delay-SLO constraint:

$$t_f = T_0 + \sum_{v \in p_f} \left( \frac{1}{x_v} + T_{v,0} + T_{v,1} b_v \right) \leq D_f \quad \forall f \in F .$$

(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:

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

(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:

$$1 \leq b_v \leq B, \quad x_v \geq 0 \quad \forall v \in V .$$

(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 $v$ 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 of optimal weighted-fair queuing data-flow-graph batch-scheduling, now with only the batch-scheduling rates $x_v$ as variables:

$$\min \sum_{v \in V} x_v(T_{v,0} + T_{v,1} R_v / x_v)$$

(8)

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

(9)

$$R_v / B \leq x_v \leq R_v \quad \forall v \in V \quad (10)$$

**Thesis 1.3.** [C2] I have shown that the optimal WFQ (weighted-fair-queuing) data flow graph batch-scheduling is unique and it can be obtained in polynomial time.

**Proof.** The mathematical program is convex. 

\[\square\]
Run-to-completion (RTC)
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 [16], and even running the scheduler itself may incur non-trivial runtime overhead. Run-to-completion execution, on the other hand, eliminates the WFQ scheduler altogether, by tracing the entire input batch though the data flow graph in one shot, by upstream modules automatically scheduling downstream modules whenever there is work to be done. This yields a simple “schedulerless” design without the risk of head-of-line blocking and internal packet drops. Our second batch-scheduler will therefore adopt run-to-completion execution.

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, 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, originally conceived in RFC 896 [17] 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.

We call the attention to some important design decisions. First, we intentionally define the trigger in the batch-size domain and not as a perhaps more intuitive timeout, since timeouts would re-introduce an WFQ 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 $b$ as this way we can tightly control downstream batch size.

What remains is to rewrite the optimization model (8)–(10) from WFQ module execution rates $x_v$ to fractional buffer triggers. Interestingly, jumping from rate-based scheduling to the run-to-completion mode is as simple 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 $x_v = \frac{R_v}{b_v}$ at batch size $b_v$. Substituting this into the optimization problem (4)–(7) yields the optimal run-to-completion data-flow-graph batch-scheduling with
variables $b_v : v \in V$:

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

(11)

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

(12)

$$
1 \leq b_v \leq B \quad \forall v \in V
$$

(13)

Thesis 1.4. [C2] I have shown that the optimal run-to-completion data flow graph batch-scheduling is unique and it can be obtained in polynomial time.

Proof. The mathematical program is convex.

Equivalence of Optimal Batch-scheduling in WFQ and RTC
The optimal data-flow-graph batch-scheduling can be expressed in our model for both explicit and run-to-completion schedule. Explicit (WFQ) schedule conforms SLOs, meanwhile a run-to-completion schedule has minimal overhead. The conversion from a feasible WFQ scheduling to RTC scheduling minimizes the scheduler overhead while keeps conforming SLOs. Thesis 1.5 shows this is doable.

Thesis 1.5. [C2] I have shown that, under assumptions A1, A2, A3, and A4, any system state feasible under the WFQ scheduling model is also attainable with a run-to-completion schedule.

The substitution $x_v = \frac{R_v}{b_v}$ enables switching between the formulations of the two batch-scheduling problem. Consequently, any WFQ scheduling problem (8)–(10) can be expressed as an equivalent set of fractional buffer triggers $b_v : v \in V$ in the run-to-completion problem (11)–(13) and vice versa.

Note, however, that the assumptions are important for the equivalence to hold; in particular, WFQ scheduling allows lossy operation (if a module is not scheduled frequently enough to keep the backlog bounded then the input queue will overflow, leading to a packet loss), whereas a RTC schedule is lossless by nature. Assumption A3 Feasibility guarantees a schedule with no packet loss, hence the equivalence in Thesis 1.5 holds.

4.2 Implementing Optimal Batch-Scheduling with Service Level Objectives
Thesis 2 presents a controller framework that implements optimal data-flow-graph batch-scheduling on a real software switches and relaxes the assumptions of the model.

Thesis 2. [C2] I have designed a one-step receding horizon optimal controller to implement the optimal batch-schedule under delay constraints in the run-to-completion model and I deployed the controller in a real-life packet processing engine. My design includes two heuristic algorithms to adaptively compute the optimal queuing; a simplified gradient optimization algorithm that is shown to be optimal in steady state, and a much simplified on-off control algorithm which does not require fractional buffer. I improved the performance of these control algorithms by introducing a heuristic to
adaptively insert/remove buffers. My implementation also contains a feasibility-recovery algorithm and a pipeline decomposition heuristics that operate on different timescales to recover the pipeline from a state when delay constraints are temporarily violated.

4.2.1 Run-to-completion Batch-scheduling Controller

Fractional buffer
Inspired by the Nagle algorithm, the controller framework focuses on the queue backlogs. This requires a queue abstraction that allows us to control queue backlogs at a fine granularity. The fractional buffer abstraction solves this issue, and its implementation is the first building block of the controller framework. Our fractional buffer implementation enables setting its size atomically and without packet loss. This allows us to set proper ingress queue sizes of modules (recall Figure 7) to achieve optimal batch-scheduling of the data flow graph and to conform to delay-SLOs. We implement the fractional buffer as a BESS module. The implementation is available at [C2].

Thesis 2.1. [C2] I have introduced the fractional buffer queuing abstraction, which enables to control the size of the batches a packet processing function receives at a fine grain.

Control Algorithms
Gradient Optimization Algorithm. A naïve 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. Such divergences may occur because a critical model assumption fails; e.g., a flow rate that we assumed constant changes or a delay-SLO gets reconfigured on the fly. To tackle this difficulty, we chose a one-step receding-horizon control framework. 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 stopped after the first iteration. This results in a coarse-grain control action, which is 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; for instance, if the A4 Static Flow Rate assumption fails and the offered load suddenly changes, the system is still able to react after at most one control period.

The controller uses the following system variables measured in each control period: mean execution rate $\bar{x}_v$ (the overbar tilde notation is to distinguish measured parameters), the mean packet rate $\bar{r}_v$, and the average batch size $\bar{b}_v$ measured at the input of the ingress buffer for each module $v \in V$ (note that $\bar{b}_v$ is distinct from the batch size at the output of the buffer, which is exactly the trigger $b_v$), plus the 95-th percentile packet delay $\bar{t}_f$ and mean packet rate $\bar{r}_f$ measured at the egress of each flow $f \in F$.

Thesis 2.2. [C2] I designed a gradient optimization algorithm to tune fractional buffer sizes. I have shown that this control algorithm attains the optimal batch-schedule in steady state.
Below, we discuss the high-level ideas in the projected gradient algorithm. The algorithm uses the method of Rosen, which searches a direction and a step size to improve the objective function value while maintaining feasibility. This is an iterative method, which cuts through the feasible region instead of jumping from vertex to vertex (e.g., simplex method); thus convergences faster but requires more computation at each step than the simplex method.

Next, we apply the projected gradient method to solve the optimization problem (11)–(13). First, we compute the objective function gradient \( \nabla l = \frac{\partial l}{\partial b_v} : v \in V \), 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:

\[
\frac{\partial l}{\partial b_v} = -\tilde{r}_v T_{0,v} = -\tilde{\lambda}_v T_{0,v}.
\]

The delay-gradients \( \nabla t_f = [\frac{\partial t}{\partial b_v} : f \in F] \) are as follows:

\[
\frac{\partial t_f}{\partial b_v} = \begin{cases} 
\frac{1}{r_v} + T_{1,v} & \text{if } v \in p_f \\
0 & \text{otherwise}
\end{cases}
\]

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 [18]. Crucially, the same calculations will also provide the Lagrangean multipliers in each iteration, to detect cases when it reached an optimal KKT (Karush–Kuhn–Tucker) point, at which point no adjustments are made to the system.

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 \( v \):

\[
\lambda_v = \min_{f \in F, v \in p_f} \frac{D_f - \tilde{t}_f}{\Delta b_v}.
\]

Finally, take \( \lambda = \min_{v \in V} \lambda_v \) and adjust the trigger of each module \( v \) to \( b_v + \lceil \lambda \Delta b_v \rceil \). Rounding the trigger up to the nearest integer yields a more aggressive controller, which will strive to maximize batch-efficiency at the cost of minor SLO violations.

**Inserting/Short-circuiting Buffers.** An unnecessary buffer on the packet-processing fast path introduces considerable delay (namely \( \frac{1}{x_v} \), which may be significant for slow-rate flows) and incurs nontrivial runtime overhead. Thesis 2.3 presents a heuristic to remove unnecessary buffers.

**Thesis 2.3.** [C2] I have designed an algorithm to adaptively remove buffers from the input of specific modules if (1) if it already receives large enough batches at the input, (2) if it would further fragment batches instead of reconstructing them, or (3) introducing the buffer already violates the delay-SLO.
Feasibility-recovery Method. The projected gradient control algorithm cannot by itself recover from an infeasible (SLO-violating) state, which may occur due to packet rate fluctuation or an overly aggressive control action. The controller contains finely tuned heuristics to recover from such states. This not only allows the controller to correct for overly eager control actions, but also serves as our main tool for rate- and SLO-adaptation: an abrupt change in the offered packet rate or SLO will most probably lead to an SLO violation, which will then trigger the feasibility-recovery process and thereby let the controller adapt to the new state rapidly.

Thesis 2.4. [C2] I have developed a feasibility recovery algorithm to bring the controller back to the feasible region.

A flow $f$ is in SLO-violation if $\hat{t}_f \geq (1 - \delta)D_f$ 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. Setting $\delta = 0.2$ on the other hand yields a controller that would start recovery well ahead of time, when a flow’s delay reaches 80% of its SLO, favoring compliance over efficiency.

Whenever a flow $f \in F$ is in SLO violation and there is a module $v$ in the path of $f$ set to a non-zero trigger ($b_v > \hat{b}_{vn}$), we attempt to reduce $b_v$ by $\left\lceil \frac{D_f - \hat{t}_f}{\partial b_v / \partial t_f} \right\rceil$. If possible, this would cause $f$ to immediately recover from the SLO-violation. Otherwise, it is possible that the invariant $b_v \geq \hat{b}_{vn}$ 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 $p_f$.

When a worker is over provisioned and the turnaround time grows beyond the SLO for a delay-sensitive flow, removing buffers is not sufficient for restoring feasibility – because in this situation turnaround time dominates delays. To handle such cases, the controller includes a pipeline decomposition method as presented in Thesis 2.5.

Thesis 2.5. [C2] I have developed a pipeline decomposition method to reallocate CPU resources across different flows by decomposing the pipeline to multiple disjunct connected worker subgraphs.

The recovery mechanism uses the analytical model to detect such cases: a worker $w$ is in inherent delay-SLO violation if a flow exists for which (14) holds, using the conservative estimate of (3).

$$\exists f \in F : \sum_{v \in V_w} (T_{v,0} + T_{v,1}B) \geq D_f \tag{14}$$

Then a flow migration process begins: 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 [10].

On-Off Control Algorithm. The one-step receding-horizon controller with the projected gradient algorithm and feasibility-recovery mechanisms is a powerful combination (for evaluation results, see Thesis 2.7). But, it requires a fractional buffers. Although, we present our fractional buffer implementation, it is not available in packet processing engines (e.g., FastClick, VPP). The on-off control algorithm (Thesis 2.6) overcomes the dependency of fractional buffers.
Thesis 2.6. \[C2\] I have designed and implemented an on-off control algorithm which switches between two extreme control actions based on the delay-SLOs: it either inserts a full batch-size buffer at the input of each module or removes queuing all together. The on-off controller does not require a fractional buffer.

The on-off control algorithm which applies bang-bang control by altering between two extremes: 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/short-circuiting and feasibility-recovery heuristics. This can be achieved by using simple buffers or queues; these modules are usually available in software switches. This way the controller works on such software switches.

Integration with BESS

The controller framework is implemented on top of BESS [11] in roughly 6,000 lines of Python/BESS code as Figure 8 shows. The controller is a standalone Python process communicating with the BESS daemon executing the data flow graph via the gRPC-based pybess API. The framework has its own BESS-inspired configuration scripts for describing the data flow graph, flows, and flow SLOs.

The initialization step involves instrumentation of the pipeline by injecting BatchMeasure modules on flow egresses to measure flow characteristics, adding FractionalBuffers to modules, hooking up I/O (e.g., starting the built-in traffic generator), and so on. During execution, the controller works on a tight-loop: measures performance metrics, calculates control, and applies trigger changes on fractional buffers. Meanwhile the controller framework keeps detailed logs about execution which can be then converted to graphs or to tabular data.

The implementation also contains new BESS module definitions: FractionalBuffer (fractional buffer) and BatchMeasure (batch statistics measure module). The source code with configuration scripts is available at \[C2\].
4.2.2 Evaluation

To understand the performance and latency-related impacts of batch-scheduling control, we implemented two baseline control algorithms (Null and Max) alongside the presented On-off and Projected Gradient algorithms: the Null algorithm performs no batch de-fragmentation at all by short-circuiting all module ingress buffers \((b_v = 0 : v \in V)\), while the Max algorithm reconstructs batches in full at the ingress buffer of all modules \((b_v = B : v \in V)\). Both baseline algorithms ignore delay-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 the controller to NFVnice, a scheduling framework originally defined for the NFV context [16]. 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.

**Thesis 2.7. [C2]** In comprehensive evaluations on 6 real-life reference scenarios I have shown that my optimal batch-scheduling controller implementation provides up to \(2\text{--}3 \times\) performance improvement while closely conforming to SLO requirements. In addition, I gave a comprehensive experimental evaluation of the operational regime where the controller provides the highest performance improvement.

The evaluations are performed on 5 representative use cases presented in Section 3.1. 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. Flows are known a priori, and pipelines are pre-configured to handle all flows. For the L2/L3 pipeline, we repeated the tests with a real traffic trace taken from a CAIDA Anonymized Internet Traces data set [19]; 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.

Static results of batch-scheduling constant bit-rate flows over 5G NFV configurations are presented in the dissertation. Results show that the presented control algorithms improve performance: Projected Gradient proved to be efficient in all cases, while On-off is too coarse-grained to exploit the full potential of batch de-fragmentation.

Next, we evaluated control algorithms under widely fluctuating system load to get a better understanding of the control dynamics. We conducted measurements to understand the system dynamics under changing packet rates and under changing delay SLOs. In both cases, the controller is able to promptly react to “bad news” such as SLO-reductions, and to “good news” such as increasing SLOs. Thanks to the careful control, the system is able to deliver \(2\text{--}3 \times\) the total throughput of the Null algorithm.

The evaluation also includes demonstrating pipeline decomposition and measuring controller overhead, but results are presented in the dissertation. Next, we attempt to obtain a general understanding of the efficiency improvements attainable with our controller. For this, we first define a set of meta-parameters that abstract away the most important factors that shape the efficiency and overhead of the controller, and then we conduct extensive evaluations in the configuration space of these meta-parameters. We abstract packet processing 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 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 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 the controller, Figure 9 shows two contour plots; the first one characterizes the speedup with Projected Gradient 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 the controller: 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 Figure 6) 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, our controller consistently brings over 1.2× improvement and never worsens performance: for workloads that do not benefit from batch de-fragmentation the controller rapidly removes useless buffers and falls back to default, unbuffered forwarding.

4.3 Resilient Data-Flow-Graph Embedding

To provide performance matching with fixed function devices, the packet processing pipeline implemented by the programmable software switch needs to be carefully mapped to the underlying hardware. The task of data-flow-graph embedding in this context is concerned with assigning each packet processing module of the data flow graph to a worker, i.e., CPU core, in a way as
to guarantee 3 crucial objectives: feasibility, efficiency, and resiliency. Each module requires a certain amount of CPU cycles to process a packet; accordingly, a feasible embedding avoids assigning too many modules to a worker, in order to preclude CPU overloads that may result in significant service degradation. A common performance bottleneck of multi-core systems is the huge toll on inter-CPU-core links [9, 10], therefore, an efficient data-flow-graph embedding is one that minimizes the number of CPU–CPU crosslinks along service chains (or crossings for short). A motivating example for minimizing crossings is shown in the left column of Figure 10.

Since even a single CPU failure can lead to a service outage by breaking the connectivity of the data flow graph, a resilient data-flow-graph embedding method will ensure that the critical packet processing paths in the software switch are immune to CPU failures. The key idea is to separate the dataflow graph to high-availability modules, which have the property that the packet processing path of at least one high-availability flow/service chain uses the module, and the rest of the modules that serve bulk flows only, and then provide resilience only for these high-availability modules by duplicating the corresponding paths in the dataflow graph. This way we can save remarkable amount of resources as compared to naively duplicating the entire dataflow graph. A motivating example for resilient embedding is shown in Figure 2.

**Thesis 3.** [C3] I have designed a novel, comprehensive analytical model for packet processing pipeline resource allocation (embedding). I have designed a method to protect high-availability data flow graph flows against CPU errors. I have formulated the embedding problem as an Integer Linear Program to find an optimal solution, and gave heuristic algorithms to obtain feasible solutions in polynomial time.
4.3.1 Data Flow Graph Embedding Model

Theorem 3.1. [C3] I have designed a model to express the resilience requirements on data flow graph flows. The model enables marking high-availability flows which requires resilience against CPU errors. Additionally, I have designed a method to protect the high-availability flows against CPU errors.

The data flow graph is given as a directed graph \( G = (V, E) \), with \( v \in V \) being the packet processing modules (i.e., L3 lookup, TTL decrement, checksum compute) and \((u, v) \in E\) the ingate-outgate connections between modules. CPUs come with fix capacity \( C \in \mathbb{N}^+ \) and in limited quantity \( n \in \mathbb{N}^+ \); we address CPUs by their ids in \([1, n]\). Modules have different CPU requirements; e.g., processing a packet through an encryption/decryption function is more costly in terms of CPU cycles than through an L2 lookup table or an IPv4 TTL decrement module. Accordingly, the CPU requirements of modules \( v \in V \) are represented by weights \( w_v \in \mathbb{N}^+ \).

Finally, a flow, or a service chain, \( f \in F \) represents a path \( p_f \) across the data flow graph \( G \).

An embedding is an assignment of each module \( v \in V \) to exactly one CPU in \( 1, \ldots, n \). The measure of the embedding is the number of crossings; here, a crossing occurs when for an edge \((u, v) \in E\) along the processing path of a flow, module \( u \) is assigned to a different CPU then module \( v \). This is captured by the following indicator function:

\[
\phi(u, v) = \begin{cases} 
1 & \text{if } u \text{ and } v \text{ are assigned to different CPUs} \\
0 & \text{otherwise}
\end{cases}
\]  

First, we define the data-flow-graph embedding with minimum crossings problem, where the concerns are feasibility and performance:

**Definition 1.** \( \text{MinSumEmbed}(G, w, F, C, n, m) \): is there an assignment of the modules of the data flow graph \( G \) to at most \( n \) CPU cores of capacity \( C \), so that (i) the sum of the weight of the modules assigned to a CPU does not exceed its capacity \( C \) and (ii) the total number of CPU–CPU crossings is at most \( m \)?

Our resilient embeddings rest on the following observation. Given a data flow graph and the set of high-availability flows as directed processing paths through the graph, we can define the set of conflicts between modules, with the understanding that conflicting modules must not be packed to the same CPU. In this setting, achieving resilience against a single CPU failure boils down to (i) duplicate the modules of high-availability flows, (ii) set conflicts between the copies, and (iii) embed the graph so that conflicting modules are never placed to the same CPU. For instance, consider the third embedding in the right column (III) of Figure 10: if any of the two CPUs fail, a functional path for the high-availability flow (flow1) still exists. Accordingly, some flows correspond to critical traffic; in the below these will be called the high-availability flows. Module conflicts are expressed by graph \( H(V, E) \). This graph is undirected; if \((u, v) \in E\) then module \( u \) is in “conflict” with module \( v \), i.e., it cannot be packed to the same CPU. Based on this discussion, we formulate the resilient data-flow-graph embedding with minimum crossings problem as a problem to find a feasible and efficient embedding but, at the same time, also satisfying the resilience constraints, expressed by the graph of conflicts \( H \).
**Definition 2.** \( \text{ResMinSumEmbed} (G, w, \mathcal{H}, \mathcal{F}, C, n, m) \): is there an assignment of the modules of the data flow graph \( G \) to at most \( n \) CPU cores of capacity \( C \), so that (i) the sum of the weight of the modules assigned to a CPU does not exceed its capacity \( C \), (ii) the total number of CPU–CPU crossings is at most \( m \), and (iii) for any two modules \( u \) and \( v \) assigned to the same CPU there is no conflict edge in \( \mathcal{H} \)?

This approach is easy to extend to \( N \) CPU failures by duplicating the modules \( N \)-times and setting a conflict between each pair of the copies. With no further constraint the conflict specification generates crosslinks by spreading sequential modules of high-availability flows among CPU cores. As an example, the right column of Figure 10 presents different resilient embeddings of two conflicting flows on 2 CPUs. Note that the required number of crossings significantly differ depending on the conflicts specified.

In the above versions of the data-flow-graph embedding problem the main concern is to minimize the total number of crossings. However, this version may allow all crossings to occur along the processing path of a single flow, unnecessarily and unfairly degrading the performance of that flow. Correspondingly, we define another version \( \text{MinMaxEmbed} \) and, respectively, \( \text{ResMinMaxEmbed} \), for the embedding problem without and with resilience constraints, where the objective is to minimize the maximum number of crossings that may occur along a single path. Our complexity characterizations and algorithms will cover all 4 versions of the embedding problem.

### 4.3.2 Complexity Analysis

**Thesis 3.2.** [C3] I have shown that both \( \text{MinSumEmbed} \) and \( \text{MinMaxEmbed} \) are NP-complete.

**Proof.** First, we prove that \( \text{MinSumEmbed} \) is in NP. A certificate is an assignment of modules to CPUs. Then, one can check whether the certificate is valid by calculating the number of crossings using the indicator function (15) and comparing the calculated value to \( m \). This can be done in polynomial time.

Second, we show that the embedding problem is NP-hard, by proving that it reduces to minimum bin-packing. A minimum bin-packing instance \( \text{MinBinPacking}(U, w, B, k) \) is given by a finite set \( U \) of items, a size \( w(u) \in \mathbb{N} \) for each \( u \in U \), and a positive integer bin capacity \( B \) [20], the output is a partition of \( U \) into disjoint sets \( U_1, U_2, \ldots, U_k \) such that the sum of the items in each \( U_i \) is \( B \) or less: \( \sum_{u \in U_i} u \leq B \ \forall i \in [1, k] \), and the measure is the number of used bins, i.e., the number of disjoint sets \( k \). Then, given a bin-packing instance \( \text{MinBinPacking}(U, s, B, k, \infty) \), it is easy to see that the data-flow-graph embedding instance \( \text{MinSumEmbed} (G(U, \emptyset), s, B, k, \infty) \) evaluates to true if and only if the bin-packing instance evaluates to true.

Finally, since the bin-packing instance does not account for minimizing the number of crossings (i.e., the number of crossings is set to \( \infty \) in the reduction), the embedding problem is NP-hard both for the total (\( \text{MinSumEmbed} \)) and the maximum (\( \text{MinMaxEmbed} \)) objective functions.

**Thesis 3.3.** [C3] I have shown that both \( \text{ResMinSumEmbed} \) and \( \text{ResMinMaxEmbed} \) are NP-complete.

**Proof.** First, it is easy to see that the problem is in NP, as again an assignment of modules to CPUs is a certificate. The resilience constraint can be checked by evaluating the indicator function (15) for each edge of the conflict graph and checking \( \forall (u, v) \in H : \phi(u, v) = 1 \).
Second, the problem is also NP-hard since it contains $\text{MinSumEmbed}(G, w, F, C, n, m)$: given an instance $\text{ResMinSumEmbed}(G, w, H, F, C, n, m)$, the instance $\text{ResMinSumEmbed}(G, w, F, C, n, m)$ evaluates to true if and only if the $\text{MinSumEmbed}$ instance evaluates to true.

The above complexity characterizations rest on the observation that the data-flow-graph embedding problem comprises an intrinsic bin-packing problem instance; that is, just deciding whether or not a data flow-graph with “weighted” nodes can be embedded into a given number of CPUs without crosslink minimization is already NP-hard, since this problem is in essence an instance of bin-packing. Interestingly, the problem remains NP-hard even if we relax the “bin-packing” constraints, i.e., we let all modules to be of uniform weight, since in this case data-flow-graph embedding maps to the balanced graph partitioning problem [21], another well-known NP-hard problem. This holds for both versions, with or without resilience constraints, and both to the sum-of-weights and the max-weight objective functions.

4.3.3 Exact Algorithm

We present an Integer Linear Program (ILP) to solve the data-flow-graph embedding with minimum crossings problem optimally and then we extend this to resilience constraints as well.

Thesis 3.4. [C3] I have formulated the feasible embedding and resilient embedding as decision problems. I have shown optimal solution can be found with exponential time complexity.

Optimal Embedding

Consider the below integer linear program (ILP).

Variables. Binary variables $x_{vi} : v \in V, i \in N$, where $x_{vi}$ takes the value 1 if module $v \in V$ is assigned to CPU $i \in N$, and zero otherwise.

Constraints. Constraint (16) sets the value of the crossing indicator function 15 $\phi$ for all edges of the data flow graph:

$$\phi(u, v) \geq x_{ui} - x_{vi} \quad \forall (u, v) \in E, \forall i \in N .$$

Constraint (17) limits the cumulative load of the modules embedded to a CPU core as the sum of modules’ weight:

$$\sum_{v \in V} w_{v} x_{vi} \leq C \quad \forall i \in N .$$

Constraint (18) ensures that each module is assigned to exactly one CPU.

$$\sum_{i \in N} x_{vi} = 1 \quad \forall v \in V$$

Finally, Constraint (19) bounds $x_{ij}$ to binary values:

$$x_{vi} \in \{0, 1\} \quad \forall v \in V, \forall i \in N .$$

Objective function(s). First, we present the objective function for the case when the goal is to minimize the total number of crossings. Consider the objective function (20):

$$\min \sum_{f \in F} \sum_{(u,v) \in \mathcal{P}_{f}} \phi(u, v) .$$
The case when the objective is the maximum number of crossings across each flow is a bit tricky. We introduce an ancillary variable $\alpha$ which bounds the number of crossings along each flow and then we minimize this variable:

$$\min \alpha : \alpha \geq \sum_{(u,v) \in p_f} \phi(u,v) \quad \forall f \in F .$$

(21)

**Optimal Resilient Embedding**

Resilience of the optimal embedding can be achieved by first formulating module conflicts as a conflict graph $H$ and then adding an extra constraint to the ILP to express conflicts as follows. **Constructing the conflict graph.** On the level of modules, the conflicts form a complete $k$-partite graph in which the vertex sets are the modules of the individual flows and edges connect each module to all modules that belong to one of the duplicates of a flow that traverses the module. To construct such module conflict graphs based on flow conflicts we present Algorithm 1 which iteratively connects every module of each flow to minimize the number of required crosslinks.

**Algorithm 1 Constructing Conflicts for Resilient Flows.**

```plaintext
procedure ConstructConflicts(resilient_flows)
    conflicts = list(module, module)
    for idx = 0; idx < resilient_flows.size(); idx++ do
        cur_flow = resilient_flows[idx]
        other_flows = resilient_flows[idx:]
        for module $\in$ cur_flow.modules do
            for other_flow $\in$ other_flows do
                for other_module $\in$ other_flow.modules do
                    if module $\neq$ other_module then
                        conflicts.append((module, other_module))
    return conflicts
```

**Resilience constraint.** Constraint (22) enforces the embedding of conflicting modules to different CPUs.

$$x_{ui} + x_{vi} \leq 1 \quad \forall (u, v) \in e(H)$$

(22)

The idea above is that whenever there is a conflict between module $u$ and $v$ then these cannot be assigned to the same CPU, say, CPU $i$, since it would result $x_{ui} + x_{vi} = 1 + 1 \leq 1$ for $i$.

**4.3.4 Heuristic Approach and Evaluation**

**Thesis 3.5. [C3]** I have developed a family of fast and effective heuristics for finding feasible embedding and resilient embedding in polynomial time. I have conducted extensive numerical evaluation and a real-life case study, and showed that the best-fit heuristic provides the best approximation of the optimal embedding in real-life applications.

The ILP may not terminate in polynomial time. Heuristic approaches on the other hand can provide solution for the embedding problems in tolerable time. For this purpose, three common algorithms were tailored. Solutions of these heuristic algorithms are either a valid embedding (every module is assigned to a CPU core) or a signal of in-feasibility. The embedding algorithms are as follows.
Worst-case embedding. The result produced by this algorithm will be used in our evaluations as a “heuristic worst-case”, whereby we attempt to maximize the number of crossings in the embedding. Accordingly, CPUs are assigned to modules in a round-robin fashion.

Executing the algorithm without conflicts, it distributes load among CPUs roughly equally (useful side effect). With conflicts, the next available CPU that runs no conflicting modules and has enough free capacity is chosen at each iteration. As a consequence, the index increment varies and CPUs load deviate.

Straw-man embedding. A random embedding serves as a straw-man proposal in our evaluations. The algorithm embeds modules one-by-one: it takes modules in sequence, builds the set of available CPUs for the module, and then it chooses one CPU from this list randomly. The choice is based on uniform distribution. A CPU is available if it is able to run the module, i.e., it has enough free capacity and runs no conflicting modules.

Best-fit embedding. We propose an embedding heuristic based on the well-known best fit decreasing algorithm for minimum bin-packing. The presented problems differ from bin packing problem in a sense of the fixed number of available bins. The algorithm embeds modules in the decreasing order of weight. At each iteration, it sorts CPUs according to their free capacity and then chooses the first available CPU running no conflicting modules.

Evaluation

The evaluation was performed on a mobile gateway packet-processing pipeline presented in 3.1. In our evaluations only bearer0 (both uplink and downlink) represents high-availability mobile voice and multimedia traffic with firm resilience requirements (QoS nodes in Figure 5). The evaluated methods are: the 3 heuristics, and ILP algorithms (ILP-sum: ILP with sum-of-crossings objective function, and ILP-max: ILP with max-crossings objective function). The evaluation focused on two scenarios: (i) embedding using minimum number of crossings without conflicts, and (ii) embedding with resilience constraints. For brevity, we detail the latter in this booklet.

The first evaluation was a numerical evaluation. The resilient embeddings took place on the an MGW pipeline with 2 bearers, 10 users from which 1 is a bearer0 user. This configuration, with the duplicate flows, has 24 flows from which 2 flows are high-availability flows (bearer0). The conflict graph had edges between the modules of the extra flows and the original flows pairwise. For embedding 10 CPUs with capacity of 200 were available.

Table 1 presents the statistics. The ILPs generally produce only a handful of crossings for each flow ($\leq 2$). Naïve implementations reside on higher values: Worst-case 10–13, Straw-man 7–12. In contrast, the Best-fit heuristics approximates the optimal solution closely, producing 0–6 crosslinks per flow at most with a $\mu$sec-scale execution time.

The second evaluation was a resiliency analysis of the embedding methods. In this experiment we conducted a case study in which we emulated a single processor core outage on a subcomponent of the MGW pipeline. We found naïve algorithms (Worst-case, Single-core Embedding) cannot provide resilience against a core outage, but embeddings based on ILP-sum and Best-fit algorithm are resilient against CPU failures.

In conclusion, the Best-fit heuristic combines the efficiency of the ILP with the short execution time of naïve heuristics, making it an efficient method for data flow graph embedding on software switches.
Table 1: Crosslink Statistics: MGW pipeline with 2 bearers, 10 CPUs of capacity of 200, and 10 users, from which one user requests a flow on bearer0 resilient to a single CPU failure.

<table>
<thead>
<tr>
<th>Embedding</th>
<th>Sum</th>
<th>Crossings per Flow</th>
<th>Exec. Time</th>
<th>Case Study Approved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Mean</td>
</tr>
<tr>
<td>Worst-case</td>
<td>282</td>
<td>10</td>
<td>13</td>
<td>11.75</td>
</tr>
<tr>
<td>Straw-man</td>
<td>208</td>
<td>7</td>
<td>12</td>
<td>8.67</td>
</tr>
<tr>
<td>Best-fit</td>
<td>59</td>
<td>0</td>
<td>6</td>
<td>2.46</td>
</tr>
<tr>
<td>ILP-sum</td>
<td>34</td>
<td>0</td>
<td>2</td>
<td>1.33</td>
</tr>
<tr>
<td>ILP-max</td>
<td>48</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

5 Application of Results

We believe our results fit into not just the academic world, but also the industry.

The work presented in this dissertation was preceded by a multi-year software switch performance evaluation industrial use cases in collaboration with researchers of Eötvös Loránd University [J2].

Since then programmable software switches became building blocks of 4G and 5G architecture [1, 4, 2]. Mobile core networks based on software are already deployed. For example, the Open Mobile Evolved Core project implements the 4G/5G mobile core user plane on top of BESS, and is already deployed in many telecommunication providers’ network [3]. Applying the presented embedding solution and batch-scheduling optimization to a similar project seems to be straightforward.

Not only mobile core, but also the 5G industrial use cases may benefit from the presented solutions. These industry-targeted use cases (e.g., centralized robot motion control in factory automation) require ultra-low-latency and high-availability for successful operation [4]. Our resilient data-flow-graph embedding provides high-availability, meanwhile the data-flow-graph batch-scheduling optimization yields low-latency.

Both of these directions are under investigation with our industrial partner, Ericsson Research Hungary under research projects (Dataplane/2015-2020 and Real-time switch/2021), and in an OTKA project (135606, “Felhő alapú hálózatok megbízhatósága”). I was fortunate to collaborate with Stefan Schmid (Universität Wien) and Barath Raghavan (USC) in these projects. I was invited to visit USC-NSL in 2020 which resulted further NFV research [O1] and RFC-text disambiguation research [C1] collaborations.

Outside of the telco world, applications using the data flow graph representation include multimedia streaming [22], big data [23], machine learning [24], and beyond. These application can benefit from our results due to the generality of data flow graph representation.
References


Publications

International Journals


International Conferences


[C4] István Pelle, Tamás Lévai, Felicián Németh, and András Gulyás, “One tool to rule them all: a modular troubleshooting framework for SDN (and other) networks”, 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR), 2015. (0.75p)


Other publications


Total Publication Score: 20