Resilient Dataflow Graph Embedding for Programmable Software Switches

Tamás Lévai
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
Budapest, Hungary
levait@tmit.bme.hu

Gábor Rétvári
MTA–BME Information Systems Research Group
retvari@tmit.bme.hu

***Abstract*—Programmable software switches have become crucial building blocks in a wide range of applications, from (virtual) data center networking to telco clouds. Modern use-cases, such as 5G, require very low latency, large throughput, and high availability. Software switches generally realize the packet processing pipeline as a dataflow graph: graph nodes correspond to simple packet processing operations and graph edges represent the control flow. The efficiency and dependability of the software switch critically depends on the way the dataflow graph is mapped to the underlying hardware resources.**

In this paper, we focus on dataflow graph embedding in a software switching context. We present an embedding approach which minimizes performance loss on inter-CPU communication across packet-processing control flow chains, and is resilient against a single CPU failure. The embedding is easy to generalize to N CPU failures. We formulate the dataflow graph embedding problem as a mathematical program, characterize the computational complexity, and we propose optimal and heuristic algorithms to solve it. The viability of our approach is confirmed in comprehensive numerical analysis on a 5G packet processing pipeline, taken from an industrial 5G NFV benchmark suite.

***Index Terms*—software switches, data flow graphs, software-defined networks, resilience**

I. Introduction

Traditional fixed-function switches and middleboxes are difficult to adapt to ever-changing applications continuously. Programmable software switches, on the other hand, allow the packet processing logics, and hence the network functions realized, to be dynamically and rapidly reconfigured in concert with the actual demands [1]. Consequently, it is expected that the 5G core network will be built on top of programmable software switches [2].

Current software switches like BESS [3], VPP [4], and FastClick [5] abstract the packet processing pipeline as a dataflow graph, in which graph nodes represent packet processing primitives and graph edges describe the control flow. Dataflow graphs provide a convenient abstraction to build essentially arbitrarily complex network functions out of simple and composable modular building blocks. Consequently, dataflow graphs are extensively used in many applications, e.g. machine learning [6], media processing [7], etc.; while we focus on the software switching use-case below, our results are easy to generalize to other applications.

In order to provide performance matching with fixed function devices, the packet processing pipeline implemented by programmable software switch needs to be carefully mapped to the underlying hardware. The task of *dataflow graph embedding* in this context is concerned with assigning each packet processing module in the pipeline to a worker, i.e., CPU core, in a way as to guarantee 3 crucial objectives: *feasibility*, *efficiency*, and *dependability*. 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 [8], [9], therefore, an *efficient* dataflow graph embedding is one that minimizes the number of CPU–CPU crosslinks along service chains (or *crossings* for short). Finally, a typical 5G use-case will run on a high-availability and resilient network service [2]. In this context, availability of a software packet processing function depends on the extent to which it is resilient to hardware failures; typical failures include CPU failures [10], [11] or the crash of the hosting virtual machine (or vCPU) [11]. Since even a single CPU failure can lead to a service outage by breaking the connectivity of the dataflow graph, a *dependable* dataflow graph embedding method will ensure that the critical packet processing paths in the software switch are immune to CPU failures.

In this paper we present novel algorithms for *feasible*, *efficient*, and *dependable* dataflow graph embedding. In particular, given a dataflow graph we seek algorithms to assign each module of the graph to a CPU core so that *(i)* no CPU cores are overloaded (feasibility), *(ii)* packet processing along each control flow path remains at a single CPU, or avoids crossing the CPU-CPU boundary as much as possible (efficiency), and *(iii)* the embedding is resilient to single-CPU failures (resilience). 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 Fig. 1.

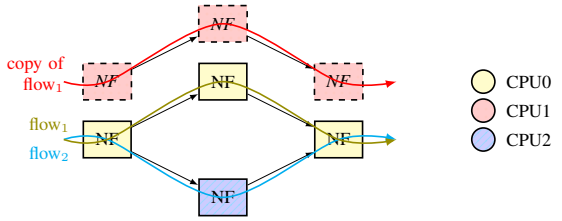


Figure 1: Resilient Dataflow Graph Embedding: initially the pipeline has 2 flows, but since flow1 is a high-availability flow it gets duplicated. The final embedding ensures if any single CPU fails, processing of flow1 traffic remains intact.

Our contributions in this paper are the following:

Models. We introduce and formulate two mathematical models for dataflow graph embedding: the *dataflow graph embedding with minimal crossings* problems asks for a feasible and efficient embedding without resilience requirements, while the *resilient dataflow graph embedding with minimal crossings* version also considers resilience to CPU failures. We introduce these problems as decision problems.

Complexity. We show that both decision problems are intractable. In fact, we are able to show that the problems are already complex in a minimalistic version when all modules have the same processing weight.

Algorithms. We formulate both dataflow graph embedding problems as Integer Linear Programs to obtain optimal solutions on smaller instances. Due to the complexity of the problems, we also need heuristics: we present simple algorithms that can provide an embedding in polynomial time.

Numerical evaluation. We demonstrate the effectiveness of our embedding approaches in extensive numerical evaluations on a mobile gateway packet processing pipeline, taken from an official 5G NFV benchmark suite [1].

Case Study. We demonstrate the effectiveness of our embedding approaches in a case study using the BESS software switch.

The rest of the paper is structured as follows. In Section II we discuss the necessary background, and then in Section III we formulate the dataflow graph embedding problem. We prove NP-hardness of the presented problems in Section IV. Section V presents algorithms for embedding, and Section VI describes our numerical analysis and case study. Finally, Section VII presents related work, and Section VIII concludes the paper.

II. Resilient Dataflow Graph Embedding

The dataflow graph is a representation for data processing programs. The graph nodes (or modules) represent processing primitives which are executed on data traversing the module, and graph edges define the control flow for data processing. Dataflow graphs are directed graphs. For instance, the dataflow graph for a simplified L3 router starts with an input rx queue, a chain of modules to check the MAC address and the `ether_type`, and then enters an L3 lookup table. The graph branches out from the L3 lookup table: for each downstream port there is a chain of group-processing modules to set the source and the destination MAC, update the TTL, recompute the checksum and send the packets to the proper tx queue.

Dataflow graph embedding involves assigning packet processing modules to workers, or processing threads, or CPU cores. Workers' processing capacity is limited, since they have resources to run only a limited number of modules. To overcome this hindrance, multiple CPU cores can be used. However, this raises a problem: whenever the processing of a packet transitions from one CPU to another (called a CPU *crossing* in this paper), the corresponding context needs to be copied and the CPU caches need to be updated, which takes a huge toll on performance and increases latency [8], [9]. Due to this trade-off, effectively increasing the performance of the software switch requires the number of crossings to be minimized, especially for performance-sensitive traffic flows.

The left column of Fig. 2 (A,B,C) presents an illustrative example. The pipeline consists of 6 modules and 2 flows and it fits into 3 CPU cores. If the pipeline is embedded as shown in panel *A* and in *B* then unnecessary crosslinks are produced. According to our preliminary measurements, this suboptimal embedding leads to significant performance penalty (see the "Delay" and "Throughput" readings in Fig. 2), whereby throughput may halve and latency may double due to the unnecessary crosslinks. The optimal solution is presented in panel *C*; this embedding yields almost 23 Mpps (million packets per second) packet rate and only about 5 μsec delay. Finding such a feasible and efficient solution is the main concern in the *dataflow graph embedding with minimal crossings* problem.

Providing resilience against CPU failures is another hard problem. Duplication of the complete dataflow graph over additional CPUs may achieve complete resilience against CPU failures. However, dataflow (sub)graph duplication involves replicating packets on the input and de-duplicating packets at the output, but replication involves significant overhead and, moreover, packet de-duplication is also a challenging technical problem [13]. Furthermore, duplicating the complete pipeline is usually unnecessary, since in a practical context only the packet-processing path of high-availability traffic needs to be resilient. An economical approach therefore is to duplicate only the modules processing high-availability traffic, e.g. voice on mobile gateway, control traffic, critical telemetry data, etc.

The right column in Fig. 1 shows a simple example: the pipeline processes 2 flows over 4 modules. If we want to provide resilience against a single CPU failure for *flow1*, all modules and edges traversed by *flow1* have to be duplicated to a new CPU core and, at the input, the packets of *flow1* have to be duplicated for all CPU cores and then de-duplicated at the egress. (We note that packet de/duplication is outside of the scope of this work, see [13].) Even if a CPU halts, the packets of the high-availability flow (*flow1*) continue to be processed without a disruption, thanks to the careful embedding. Finding such failure-tolerant embeddings is the main task in the *resilient dataflow graph embedding with minimal crossings* problem.

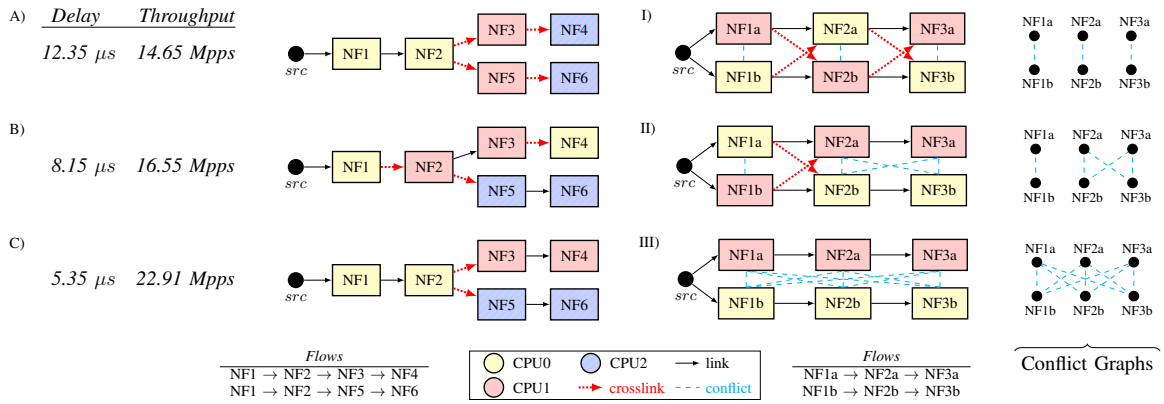


Figure 2: Dataflow Graph Embedding: *Left column:* embedding on 3 CPUs w/o resilience requirements. Each CPU has the capacity to run 2 modules. The number of crosslinks differ: A) 4, B) 3, C) 2. Additionally, the number of crosslinks across a single flow also differ: A,B) 2, C) 1. The decreasing number of crosslinks induces increasing performance according to measurements made with a BESS implementation [12]. *Right column:* embedding on 2 CPUs, each with capacity to run 3 modules, with resilience against single CPU failure. The number of crosslinks differ: I) 4, II) 2, III) 0, and crossings for a single flow are: I) 2, II) 1, III) 0, respectively.

III. Problem Formulation

The main goal of dataflow graph embedding is to assign modules to workers in a way as to guarantee feasibility, in that no CPU gets overloaded, efficiency, in that the number of CPU-CPU hops along packet processing paths is minimized, and resilience, so that critical packet processing pathways are immune to single-CPU failures. Below, we formulate the problem first without resilience constraints, and then we consider single-CPU failures as well.

The dataflow graph is given as a directed graph $D(V, A)$, with $v \in V$ being the packet processing modules (i.e., L3 lookup, TTL decrement, checksum compute) and $(u, v) \in A$ the ingate-outgate connections between modules. CPUs come with fix capacity $C \in \mathbb{N}^+$ and in limited quantity $n \in \mathbb{N}^+$. Modules have different CPU requirements; e.g. processing a packet through en encryption/decryption function is more costly in terms of CPU cycles than through an L2 lookup table or a 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 dataflow graph D.

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 $(u, v) \in A$ 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:

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

First, we define the *dataflow graph embedding with minimum crossings* problem, where the concerns are feasibility and performance:

Definition 1. MinSumEmbed *(G, w, F, C, n, m): is there an assignment of the modules of the dataflow 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 dataflow 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 of Fig. 2: 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 dataflow 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. ResMinSumEmbed *(G, w, H, F, C, n, m): is there an assignment of the modules of the dataflow 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 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 Fig. 2 presents different resilient embeddings of two conflicting flows on 2 CPUs. However, depending on the conflicts specified, the required number of crossings significantly differ.

In the above versions of the dataflow 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 MINMAXEMBED and, respectively, 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.

IV. COMPLEXITY

In this section we prove the NP-completeness of the presented decision problems.

Theorem 1. *Dataflow graph embedding with minimum crossings is NP-complete for both measures.*

Proof: First, we prove that 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 (Eq. 1) 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 MINBINPACKING(U, w, B, k) is given by a finite set U of items, a size $s(u) \in \mathbb{N}$ for each $u \in U$, and a positive integer bin capacity B [14], 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, and the measure is the number of used bins, i.e., the number of disjoint sets k. Then, given a bin-packing instance MINIMUMBINPACKING(U, s, B, k), it is easy to see that the dataflow graph embedding instance MINSUMEMBED ($G(U, \emptyset)$, s, B, k, ∞) 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 ∞ in the reduction), the embedding problem is NP-hard both for the total (MINSUMEMBED) and the maximum (MINMAXEMBED) objective functions. ■

Next, we also consider resilience requirements.

Theorem 2. *Resilient dataflow graph embedding with minimum crossings is NP-complete for both measures.*

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 (Eq. 1) for each edge of the conflict graph and checking that $\forall (u, v) \in H : \phi(u, v) = 1$.

Second, the problem is also NP-hard since it contains MINSUMEMBED (G, w, F, C, n, m): given an instance RESMINSUMEMBED (G, w, H, F, C, n, m), the instance MINSUMEMBED (G, w, F, C, n, m) evaluates to true if and only if the MINSUMEMBED instance evaluates to true. ■

The above complexity characterizations rest on the observation that the dataflow graph embedding problem comprises an intrinsic bin-packing problem instance; that is, just deciding whether or not a dataflow-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 dataflow graph embedding maps to the *balanced graph partitioning* problem [15], 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.

V. ALGORITHMS

First, we present an ILP to solve the "dataflow graph embedding with minimum crossings" problem optimally and then we extend this to resilience constraints as well. Then, various heuristics are presented to obtain "good enough" solutions in polynomial time.

A. Optimal Embedding

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 (2) sets the value of the crossing indicator function ϕ for all edges of the dataflow graph:

$$\phi(u, v) \geq x_{ui} - x_{vi} \quad \forall (u, v) \in A, \forall i \in N \ . \tag{2}$$

Constraint (3) limits the cumulative load of the modules embedded to a CPU core as the sum of modules' weight:

$$\sum_{v \in V} w_v x_{vi} \leq C \quad \forall i \in N \ . \tag{3}$$

Constraint (4) ensures that each module is assigned to exactly one CPU.

$$\sum_{i \in N} x_{vi} = 1 \quad \forall v \in V \ . \tag{4}$$

Finally, constraint (5) bounds x_{ij} to binary values:

$$x_{vi} \in \{0, 1\} \quad \forall v \in V, \forall i \in N \ . \tag{5}$$

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 (Eq. 6):

$$\min \sum_{f \in F} \sum_{(u,v) \in p_f} \phi(u,v) \ . \tag{6}$$

The case when the objective is the *maximum* number of crossings across each flow is a bit tricky. We introduce an ancillary variable α which bounds the number of crossings along each flow and then we minimize this variable:

$$\min \alpha : \alpha \geq \sum_{(u,v) \in p_f} \phi(u,v) \qquad \forall f \in F \ . \tag{7}$$

B. Optimal Resilient Embedding

Resilience of the optimal embedding can be achieved by *(i)* formulating module conflicts as a conflict graph H and then *(ii)* 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

```
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 ∈ cur_flow.modules do
            for other_flow ∈ other_flows do
                for other_module ∈ other_flow.modules do
                    if module! = other_module then
                        conflicts.append((module, other_module))
    return conflicts
```

Resilience constraint. Constraint (8) enforces the embedding of conflicting modules to different CPUs.

$$x_{ui} + x_{vi} \leq 1 \qquad \forall (u,v) \in e(H) \tag{8}$$

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 in this case we would have $x_{ui} + x_{vi} = 1 + 1 \not\leq 1$ for i.

C. Heuristics

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. All heuristics work on the decision version of the problem introduced in Section III, solutions are either a valid embedding or a signal of infeasibility.

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 (a useful side effect). With conflicts, the next available CPU that runs no conflicting modules and has enough free capacity is chosen at each iteration.

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* [16] algorithm for minimum bin-packing. The algorithm presented in Algorithm 2 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. If there is no fitting CPU, the algorithm signals infeasibility.

Algorithm 2 Best-fit Embedding with Conflicts

```
procedure BestFitEmbed(G, H, V, cpus[n])
    mapping = dict(module, cpu)
    modules_sorted = sort(V, weight, decreasing)
    for module ∈ modules_sorted do
        conflicts = get_conflicting_modules(module, G, H)
        cpus_sorted = sort(cpus, load, decreasing)
        for cpu ∈ cpus_sorted do
            if not ∃m ∈ conflicting_modules : mapping[m] = cpu then
                if cpu.load + module.weight ≤ C then
                    mapping[module] = cpu
                    break
            if cpu == cpus_sorted.last() then
                signal in-feasibility
                exit
    return mapping
```

VI. Evaluation

In this section, we evaluate our embedding algorithms on a real-life use-case.

A. Evaluation Setup

The evaluation was performed on a mobile gateway packet-processing pipeline (Fig. 3) taken from an official 5G NFV benchmark suite [1]. A 5G mobile gateway connects a set of mobile user equipments to the public Internet. This requires a complex pipeline with differentiated traffic classes (called "bearers"). Traffic flows either in uplink or downlink direction, and is further classified among bearers implementing different Quality of Service levels. Users have connections on multiple bearers both in the uplink and downlink directions and each user's connection is considered as a separate flow. In our evaluations bearer0 (both uplink and downlink) represent high-availability mobile voice and multimedia traffic with firm resilience requirements, while the rest of the bearers represent bulk traffic. The number of concurrent flows is $2\times$ the number

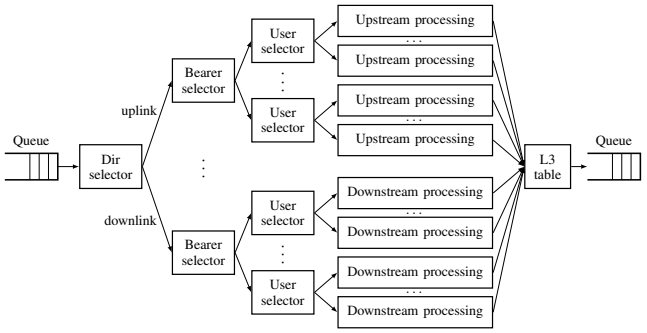


Figure 3: Mobile Gateway (MGW) Pipeline

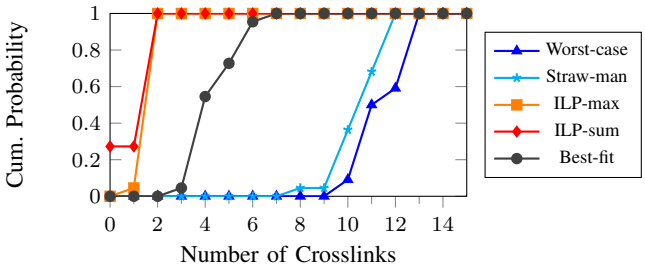


Figure 4: Distribution of Crossings across Flows: MGW pipeline with 2 bearers, 10 users, and one user on bearer0, on 10 CPUs with capacity of 200 units. Resilience constraints are not set.

of users on the first bearer (bearer0) due to separate uplink and downlink connections. The number of bearers, users, and users of the resilient bearers (bearer0 users) as well as the capacity and the number of CPUs are parameters. Module weights were set to differentiate downstream processing (weight: 10) and upstream processing (weight: 20), and the remaining functions such as splitters (weight: 1).

The evaluation focused on two scenarios: *(i)* embedding using minimum number of crossings without conflicts, and *(ii)* embedding with resilience constraints. In both scenarios the following methods were evaluated:

- **Worst-case**: approach using round-robin assignment.
- **Straw-man**: approach using random assignment.
- **Best-fit**: algorithm based on the best fit heuristic.
- **ILP-sum**: ILP with sum-of-crossings objective function.
- **ILP-max**: ILP with max-crossings objective function.

The ILP and the heuristics were implemented in C++ using the LEMON graph library and Gurobi Optimizer version 8.1.1. The MGW pipeline was synthesized to a custom Lemon Graph Format file which stores modules, links, module weights, CPU parameters, descriptions of flows, and the edges of the conflict graph. This file served as an input to our implementation. The evaluation was running on a server with 6×2.4GHz CPU (power-saving disabled) and 64GB RAM installed with the Debian GNU/Linux operating system. The source code is available on GitHub [12].

B. Results

Dataflow graph embedding with minimum crossings. In our first evaluations we study the performance of the heuristics and the ILP without resilience constraints. The number of CPUs was set to be high enough for all the embedding procedures; in particular, 10 CPUs, each with a capacity of 200 units, were available. The pipeline was provisioned with 2 bearers and 10 users, from which only 1 user requested a flow at bearer0, yielding 22 flows in total. We present the empirical cumulative density function of crosslinks across flows in Fig. 4. The ILPs using either one of the objective functions use 0–2 crosslinks per flow, the worst-case and the random methods assign 8–13 crosslinks, while the Best-fit algorithm performs close to the optimal solution, with 3–6 crosslinks per flow.

	Crossings per Flow					Exec.
Embedding	**Sum**	**Min**	**Max**	**Mean**	**Std. dev**	**time**
Worst-case	282	10	13	11.75	1.03	$143\mu s$
Straw-man	208	7	12	8.67	1.01	$272\mu s$
Best-fit	59	0	6	2.46	1.98	$199\mu s$
ILP-sum	34	0	2	1.33	0.96	$1.1min$
ILP-max	48	2	2	2	0	$0.3min$

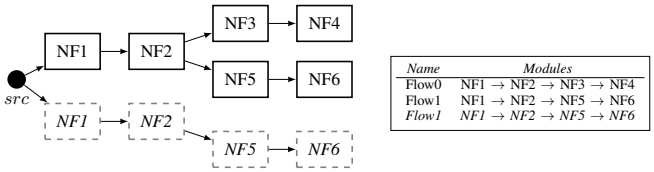
Table I: Crosslink statistics: MGW pipeline with 2 bearers, 10 CPUs of capacity of 200, and 10 users, from which one user requests a flow on bearer0 resilient to a single CPU failure.

Resilient dataflow graph embedding. Next, we compare the performance of the heuristics and the ILP on the same pipeline as above, but this time with resilience requirements. In particular, the embedding produced by the algorithms was required to be resilient to a single CPU failure on the processing path of the high-availability bearer0 flows. This configuration, with the duplicate flows, has 24 flows from which 2 flows are high-availability flows (bearer0).

Table I presents the statistics. The ILPs generally produce only a handful of crossings for each flow (≤ 2), as opposed to the heuristics which cover a wide range from 0 to 13 crossings. The Worst-case approach provides the worst result with 10–13 crosslinks, and the Straw-man algorithm also generates prohibitively many crossings (7–12). In contrast, the Best-fit heuristics approximates the optimal solution very closely, producing 0–6 crosslinks per flow at most. Apart from approximating the optimal embedding, the Best-fit algorithm's execution time varies in the same range as that of the naïve algorithms.

Scalability. The execution time is a relevant metric in this context, as the dataflow graph may need to be remapped to the underlying hardware every time the processing logic changes, e.g., whenever a new user is provisioned in the mobile gateway. This may result in the embedding algorithm being executed several times per second. To understand the scalability of the algorithms to larger use cases, we measured the processing time of the embedding functions on increasingly more complex configurations.

Increasing the number of bearer0 users from 1 to 5 and the number of CPUs from 10 to 40, and leaving the rest of the parameters intact, the processing time of ILP-max increases to



Name	*Modules*
Flow0	NF1 → NF2 → NF3 → NF4
Flow1	NF1 → NF2 → NF5 → NF6
Flow1	*NF1 → NF2 → NF5 → NF6*

Figure 5: Simplified 5G MGW pipeline taken from Fig. 2. Flow1 is required to be resilient to a single CPU failure and hence all modules Flow1 traverses are duplicated. Duplicate modules are marked dashed and italic in the figure.

11.52 minutes, and ILP-sum processing time increases to 3.62 hours. Conversely, the Best-fit method finishes in 2022 μs, while the Straw-man and Worst-case embedding algorithms run out of available CPUs and therefore do not produce a feasible embedding. Next, we consider the single-CPU-failure resilient scenario with 2 bearers and 10 users, from which 5 are bearer0 users, and CPUs with a capacity of 200 units. The Straw-man approach requires 66 CPUs, and the Worst-case approach requires 98 CPUs. On the contrary, Best-fit requires only 40 CPU cores, while both ILP-sum and ILP-max methods require only 30 cores.

As a conclusion, we see that the ILP solves the embedding problem efficiently but the usability is limited due to the large processing overhead. The naïve heuristics, on the other hand, have limited usability in the hardware resources domain: the number of CPU cores required by Straw-man, and Worst-case approaches is prohibitive. The Best-fit approach, however, scales well in time domain and resource domain as well and it provides an appealing option for practical implementations.

C. Case Study

Finally, we demonstrate the real-life applicability of our embedding algorithms in a case study using the BESS software switch. In this measurement we are focusing on *(i)* the steady-state performance and *(ii)* resilience against a single CPU failure on a running software switch.

Evaluation setup. For brevity, we used a simplified 5G MGW pipeline as presented in Fig. 5. First, we implemented the pipeline on top of BESS. In this setup, Flow1 corresponds to critical traffic. Next, we synthesized the input file to our embedding algorithms from the running pipeline. Some parameters were manually adjusted: module weights were set proportional to the number of traversing flows and conflicts were set between all copies of the modules of Flow1 (see the conflict graph construction method in Algorithm 1). Following that, we instrumented the BESS pipeline from the results of the embedding algorithms. For resiliency analysis, the CPU failure was emulated by (dis)connecting the modules provisioned to the first CPU core at the 1st second of the measurement and then re-connecting them at the 2nd second.

The evaluated methods were the following: resilient methods: *ILP-sum*, *Best-fit*, and non-resilient methods: *Worst-case* and running the pipeline on a single core (*1CPU*).

Results. We measured steady state-performance (without CPU failure) of the different embedding methods. We present

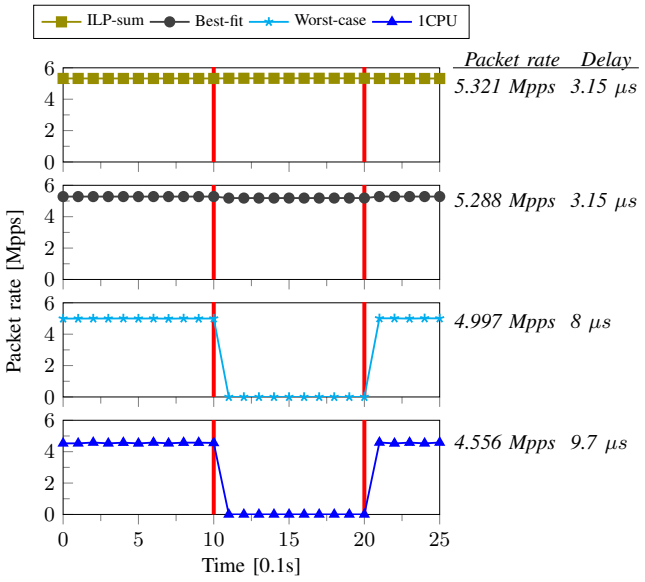


Figure 6: Throughput of Flow1 during a single CPU failure event: average steady-state throughput and delay. CPU state changes are marked by thick red lines. Note that resilient embedding methods are not affected by the CPU outage.

averaged steady-state throughput in Fig. 6. The embedding based on ILP-sum and Best-fit algorithm provides similar results, at 5.3 Mpps. Other embedding methods produce worse throughput; worst-case maxes out at 5 Mpps, and single-core embedding yields approx. 4.6 Mpps.

We analyzed CPU failure-tolerance of the embedding methods by emulating a single processor core outage, see Fig. 6. The CPU failure at the first second cuts off non-resilient embedding methods, however resilient embedding methods successfully switch over to the duplicate flow with no outage in packet processing.

To conclude, we see that the steady-state performance and resilience are indeed dependent on the used embedding method. Embedding based on ILP-sum and Best-fit algorithm are able to overcome the performance degradation factors of crosslinks and heavy CPU load, and are resilient against CPU failures.

VII. Related Work

Embedding dataflow graphs for high-performance. Existing work focuses on dataflow graph embedding in the context of software switching and VNF chaining. Metron [9] achieves line-rate performance for NF-chains by eliminating unnecessary inter-core communication. SNF [8] dissects traffic-classes of an existing NF chain, and synthesizes an equivalent high-performance NF chain by consolidating several packet processing operations (e.g., read, drop, write-once, etc.) into single NFs. No prior work as far as we know of considers resiliency constraints in high-performance dataflow graph embedding.

Network embedding and graph partitioning. Graph embedding has a broad applicability and is a recurring research

topic. Research related to our work include a bi-criteria approximation algorithm for partitioning a graph into a given number of equally sized components [17] and for roughly equal sized bins [15]. Online balanced graph partitioning algorithms for placing VMs communicating in time-varying fashion to servers dynamically are described in [18] and [19].

Virtual network embedding is a technical sub-problem of graph embedding. The virtual network is represented as a graph and has to be embedded onto a substrate graph. This problem is proven to be NP-hard [20]. Regarding our work, DOT [21], a distributed network emulator, highlights the problem of limiting physical cross-links during the embedding of the emulated network to a grid-like physical topology. Again, resiliency concerns are outside the scope of these graph-embedding proposals.

Resilient NF-chain embedding. Providing NF chain resilience against single-link and single-node failures is discussed in [22]. REINFORCE [23] is an integrated framework to provide chain-wide network function resilience keeping high performance. To the best of our knowledge we are the first to propose and evaluate resilient dataflow graph embedding methods for software switches.

VIII. Conclusions and Future Work

Programmable software switches abstract the packet processing pipeline as a dataflow graph, which needs to be assigned to a set of workers subject to feasibility, efficiency, and dependability constraints. We introduce a mathematical model to control and optimize this embedding process to yield high-performance and provide resilience against CPU failures. We proved the NP-completeness with two important objective functions, formulated the optimal embedding problem as an ILP, and developed effective and fast heuristics. Our evaluations show that naïve heuristics cannot cope with embedding problems at scale except for the best-fit heuristic. Correspondingly, we propose the best-fit heuristic to be used in practical dataflow graph embedding implementations due to its close approximation of ILP results, and its negligible execution time.

Future work is aimed at extending the model to handle delay and rate service level objectives, and improving resilience against CPU failures of best effort modules and thus increasing the existing resiliency to the complete pipeline.

Acknowledgment

We would like to thank Stefan Schmid for his valuable comments. This work was partially supported by Ericsson Research, Hungary, G.R. is also with the MTA–BME Momentum Network Softwarization Research Group.

References

[1] 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, pp. 2621–2630, Dec. 2018.

[2] NGMN Alliance, "5g white paper," *Next generation mobile networks, white paper*, pp. 1–125, 2015.

[3] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[4] E. Warnicke, "Vector Packet Processing - One Terabit Router," July 2017.

[5] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 5–16, 2015.

[6] H.-Y. Chen *et al.*, "TensorFlow: A system for large-scale machine learning," in *USENIX OSDI*, vol. 16, pp. 265–283, 2016.

[7] W. Taymans, S. Baker, A. Wingo, R. S. Bultje, and S. Kost, *GStreamer Application Development Manual*. United Kingdom: Samurai Media Limited, 2016.

[8] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: synthesizing high performance nfv service chains," *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016.

[9] 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*, pp. 171–186, 2018.

[10] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 25–36, 2017.

[11] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, "Failure analysis of virtual and physical machines: Patterns, causes and characteristics," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 1–12, 2014.

[12] "Source Code and Artifacts on GitHub." https://github.com/levaitamas/dataflow_graph_embedding.

[13] H. Sadok, M. E. M. Campista, and L. H. M. K. Costa, "A case for spraying packets in software middleboxes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pp. 127–133, 2018.

[14] P. Crescenzi and V. Kann, "A compendium of NP optimization problems," 2005 (accessed April 25, 2019). https://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html.

[15] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, pp. 929–939, Nov 2006.

[16] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for np-hard problems," in *Approximation Algorithms for NP-hard Problems* (D. S. Hochbaum, ed.), ch. Approximation Algorithms for Bin Packing: A Survey, pp. 46–93, Boston, MA, USA: PWS Publishing Co., 1997.

[17] R. Krauthgamer, J. S. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, (Philadelphia, PA, USA), pp. 942–949, Society for Industrial and Applied Mathematics, 2009.

[18] C. Avin, A. Loukas, M. Pacut, and S. Schmid, "Online balanced repartitioning," *CoRR*, vol. abs/1511.02074, 2015.

[19] M. Henzinger, S. Neumann, and S. Schmid, "Efficient distributed workload (re-)embedding," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, pp. 13:1–13:38, Mar. 2019.

[20] E. Amaldi, S. Coniglio, A. M. Koster, and M. Tieves, "On the computational complexity of the virtual network embedding problem," *Electronic Notes in Discrete Mathematics*, vol. 52, pp. 213 – 220, 2016. INOC 2015 – 7th International Network Optimization Conference.

[21] A. R. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba, "Design and management of dot: A distributed openflow testbed," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014.

[22] A. Hmaity, M. Savi, F. Musumeci, M. Tornatore, and A. Pattavina, "Virtual network function placement for resilient service chain provisioning," in *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pp. 245–252, 2016.

[23] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization based services," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pp. 41–53, 2018.