# Fast ReRoute on Programmable Switches

Marco Chiesa*, Roshan Sedar†, Gianni Antichi‡,
Michael Borokhovich§, Andrzej Kamisiński¶, Georgios Nikolaidis‖, Stefan Schmid**
*KTH Royal Institute of Technology †Telecommunications Technological Center of Catalonia
‡Queen Mary University of London §Independent Researcher ¶AGH University of Science and Technology
‖Intel, Barefoot Switch Division **University of Vienna

***Abstract*—Highly dependable communication networks usually rely on some kind of Fast Re-Route (FRR) mechanism which allows to quickly re-route traffic upon failures, entirely in the data plane. This paper studies the design of FRR mechanisms for emerging reconfigurable switches. Our main contribution is an FRR primitive for *programmable* data planes, PURR, which provides low failover latency and high switch throughput, by *avoiding packet recirculation*. PURR tolerates multiple concurrent failures and comes with minimal memory requirements, ensuring *compact* forwarding tables, by unveiling an intriguing connection to classic "string theory" (*i.e.*, stringology), and in particular, the shortest common supersequence problem. PURR is well-suited for high-speed match-action forwarding architectures (*e.g.*, PISA) and supports the implementation of a broad variety of FRR mechanisms. Our simulations and prototype implementation (on an FPGA and a Tofino switch) show that PURR improves TCAM memory occupancy by a factor of $1.5$x–$10.8$x compared to a naïve encoding when implementing state-of-the-art FRR mechanisms. PURR also improves the latency and throughput of datacenter traffic up to a factor of $2.8$x–$5.5$x and $1.2$x–$2$x, respectively, compared to approaches based on recirculating packets.**

***Index Terms*—programmable networks, network robustness, fast reroute, fast failover, P4, shortest common supersequence.**

## I. Introduction

Emerging applications, *e.g.*, in the context of business and entertainment, pose stringent requirements on the dependability and performance of the underlying communication networks, which have become a critical infrastructure of our digital society. In order to meet such requirements, many communication networks provide *Fast Re-Route* (FRR) mechanisms [1], [2], [3], which allow to quickly reroute traffic upon unexpected failures, entirely in the data plane. By proactively provisioning the switches with backup forwarding rules, the robustness and availability of a network can be increased: as soon as a switch detects a failure, *i.e.*, defective link or port, it quickly detours traffic using local backup rules.

Networking equipment manufacturers have so far integrated FRR capabilities directly in the silicon of their switches, allowing network operators to simply use such functionality as a *black-box* option. Emerging Programmable Data Planes [4], PDPs, are about to break this black-box approach to data plane network functionalities. Indeed, by allowing network operators to deploy customized packet processing algorithms, PDPs are considered a key enabler of many interesting new use cases including monitoring [5], [6], traffic load-balancing [7], and many others [8]. However, little is known today about how to implement FRR mechanisms with reconfigurable switches. One simple approach is to recirculate the packet back at the input of the switching pipeline when a failure has been detected and select a different output port. This however leads to increased packet processing latency and reduced throughput.

We therefore aim to make FRR *efficient*, thus avoiding expensive packet recirculations, and *programmable*, thus allowing operators to pick any FRR mechanism (*e.g.*, [9]). This is challenging and involves multiple goals:

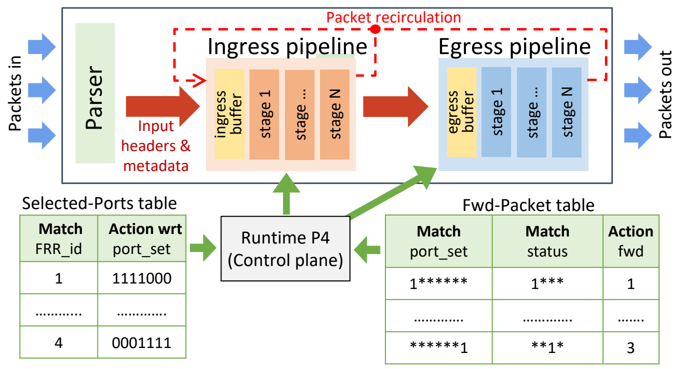
- ***Flexibility:*** We aim to devise an *FRR primitive* that supports a broad variety of FRR mechanisms robust to *single* and *multiple* link failures [10], [11]. FRR mechanisms deal with the computation of primary and backup forwarding rules.
- ***Low latency and high throughput:*** Packets affected by a failure should be rerouted to an alternate active port as fast as possible without incurring any packet processing degradation. This means packet processing latency should not depend on the number of failed ports on a switch: a key requirement for latency-critical applications.
- ***Memory efficiency:*** A programmable FRR mechanism should come with minimal memory requirements, *i.e.*, the resulting forwarding tables are required to be *compact*. Memory (especially TCAM) is, in fact, a scarce yet precious resource of today's hardware PDPs [12].

In this paper we propose a new ***FRR primitive***, PURR, that serves as a building block for implementing FRR mechanisms while meeting the above requirements. At the heart of PURR lies a technique that *avoids recirculating packets* through the switch pipeline in search of an active port, which would lead to worsened performance, *i.e.*, higher latency and lower throughput. To provide memory efficiency, PURR leverages a connection between compact FRR forwarding tables and algorithmic string theory (*i.e., stringology*): the main theoretical contribution of this paper. Specifically, we show that it is possible to implement a wide range of FRR mechanisms very efficiently using our primitive, by modeling the optimization problem as a variant of a *Shortest Common Supersequence (SCS)* problem. To this end, we devise and analyze several new algorithms to efficiently solve this SCS variant. We show how optimized SCS solutions translate into low-memory realizations of the given FRR mechanisms.

In summary, we make the following contributions:

- We explore the design space alongside the trade-offs of implementing FRR mechanisms on hardware-based PDPs.
- We propose PURR, a new FRR primitive that can be

This paper is an extended version of [?] and [?].



Selected-Ports table

| **Match** FRR_id | **Action wrt** port_set |
|---|---|
| 1 | 1111000 |
| ............ | ............. |
| 4 | 0001111 |

Fwd-Packet table

| **Match** port_set | **Match** status | **Action** fwd |
|---|---|---|
| 1****** | 1*** | 1 |
| ............. | ............. | ....... |
| ******1 | **1* | 3 |

Figure 1: PISA abstraction with PURR pipeline.

adopted as a building block for implementing FRR algorithms. PURR provides very low failover latency and high packet processing throughput by requiring a *single* TCAM lookup, and low memory overhead by exploiting an unexplored connection to classic algorithmic string theory.

- PURR comes with solid algorithmic underpinnings. In particular, we show that the underlying problem is a variant of SCS without repetitions, and prove that this variant is still $NP$-hard. We then present a novel and efficient heuristic to solve this variant of the SCS problem, which may be of interest beyond the scope of this paper.
- We report on an extensive evaluation, combining analytical results and simulations. We assessed PURR using micro-benchmarks and large-scale simulations. Our main findings show PURR dramatically *reduces memory requirements* by a factor of 1.5x–10.8x for a variety of existing FRR mechanisms compared to a naïve approach. Our large-scale simulations show that *packet recirculation has devastating effects on the flow completion times* of the latency-sensitive flows, up to 2.8x—5.5x worse than PURR.
- We assessed the feasibility of realizing PURR in practice by implementing it in P4 on the *bmv2* software switch [13], a Tofino switch [14], and an FPGA [15].
  Our code is available and fully reproducible [16].

## II. Background and Motivation

**P4 background.** P4 [4] is a programming language specifically designed to program data plane packet processing pipelines based on a match-action architecture. The P4 language is target-independent [17], *i.e.*, it abstracts from the specific hardware characteristics of a switch. A P4 compiler translates high-level P4 programs into target-dependent switch configurations. Network operators write forwarding behavior using P4 and subsequently compile these programs into P4-enabled switches using vendor-specific compilers. In this paper, we focus solely on hardware-based P4 switches.

The top part of Fig. 1 depicts a high-level abstraction of the standard de-facto P4 packet processing pipeline, *i.e.*, the PISA pipeline [17]. This pipeline consists of a *parser* component followed by an *ingress* and an *egress* forwarding pipelines. The parser can be configured by the network operators to match arbitrary (ad-hoc) fields in the packet header. Each pipeline consists of a sequence of match-action stages, similarly to OpenFlow. The network operator can decide upon the size

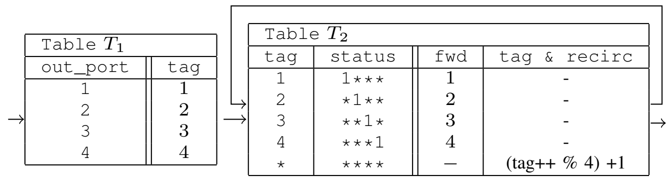


Table $T_1$

| out_port | tag |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |

Table $T_2$

| tag | status | fwd | tag & recirc |
|---|---|---|---|
| 1 | 1⋆⋆⋆ | 1 | - |
| 2 | ⋆1⋆⋆ | 2 | - |
| 3 | ⋆⋆1⋆ | 3 | - |
| 4 | ⋆⋆⋆1 | 4 | - |
| ⋆ | ⋆⋆⋆⋆ | — | (tag++ % 4) +1 |

Figure 2: A packet recirculation forwarding table.

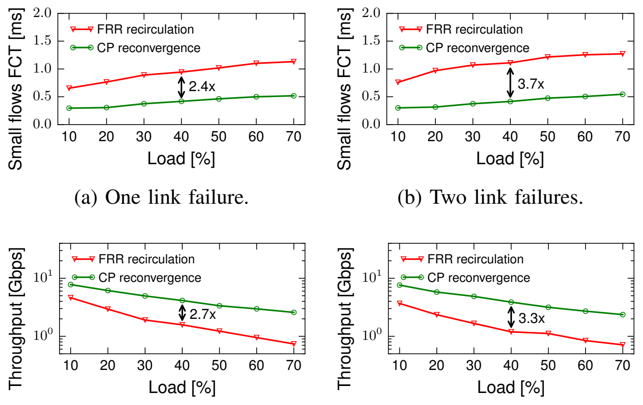
and number of match tables, their matching type (*e.g.*, exact, wildcard, range), and the actions associated with a match "hit" (*e.g.*, rewrite the packet header, increase a counter). Similarly to OpenFlow, P4 programmers can use *metadata* fields to carry information across different stages and match on those fields. The metadata attached to a packet is lost as soon as the packet leaves the switch. It is worth noting that P4 does not dictate how the match-action tables are mapped onto the TCAM and other memories contained within each stage of the pipeline. Clearly, different memories strike different trade-offs in terms of cost, energy consumption, and latency. TCAM memories *support* a wildcard, which we will leverage in the rest of the paper. The complexity of computing the mapping of the match tables to the hardware memories is left to the P4 compiler, which is different for each target packet processing switch.

**P4 and Fast ReRoute (FRR).** The P4 abstraction has gained ever-growing interests from the networking community thanks to its flexibility and general-purpose interface. Yet, P4 comes with no built-in support for commonly used Fast Re-Route (FRR) forwarding operations, *i.e.*, the forwarding action consists of a sequence of ports such that a packet matching that action is forwarded to the first active (*i.e.*, non-failed) port in the sequence. This is similar to FRR groups, henceforth called FRR *sequences*, of OpenFlow [18]. For example, consider an FRR mechanism that *i)* indexes all the switches' ports from 1 to $k$ and *ii)* when the switch fails to send a packet on a port with index $i$, it tries with ports $i+1$, $i+2$, and so on, modulo the number of ports, until an active port is found. We call the resulting FRR sequences (*i.e.*, $\langle 1,2,3,4\rangle$, $\langle 2,3,4,1\rangle$, $\langle 3,4,1,2\rangle$, and $\langle 4,1,2,3\rangle$ ), *circular* FRR sequences.

Based on our extensive discussions with P4 developers, the implementation of FRR sequences in P4 is today left to the operator [19]. We note that FRR primitives devised in different contexts (*e.g.*, BGP-PIC [20], [21]) cannot support arbitrary FRR sequences (namely, only FRR sequences of size 2).

Implementing an *FRR primitive* is far from being trivial. Without specific built-in FRR hardware support within the hardware switch devices, operators have to rely only on the match-action processing pipeline to enable quick packet forwarding recomputation upon *any* number of link failures. One way to achieve this goal entails *recirculating a packet* through the switch pipeline multiple times in search of the first non-failed port in an FRR sequence, or alternatively, by writing a P4 program that checks the state of the links in the FRR sequence either *sequentially* (*i.e.*, through multiple stages) or in *parallel* (*i.e.*, using a TCAM). We now analyze these three different possible solutions.

**FRR sequences with packet recirculation.** One simple way to implement FRR is to recirculate a packet until an active outgoing port is found. Consider the simple example shown



(a) One link failure. (b) Two link failures.

(c) One link failure. (d) Two link failures.

Figure 3: Packet recirculation performance analysis.

in Fig. 2 in which we want to support an FRR mechanism that is based on the aforementioned set of FRR circular sequences, *i.e.*, $\langle 1,2,3,4\rangle$, $\langle 2,3,4,1\rangle$, $\langle 3,4,1,2\rangle$, and $\langle 4,1,2,3\rangle$. To realize an FRR sequence with packet recirculation, we store in the packet header/metadata information about the port through which we want to forward the packet, *i.e.*, the `tag` field, and increment this value if the pointed port is down. The first table $T_1$ is used to simply attach the initial tag to a packet. Each packet carries a port `status` metadata where each bit in the `status` metadata represents the status of a port: it is set to 1 if the port is active or to 0 otherwise. We assign a port identifier to each port of the switch and let the $i^{\text{th}}$ bit in `status` represent the $i^{\text{th}}$ port of the switch. The `status` matching operation simply checks whether the port indexed by the `tag` field is up or down. For instance, consider a packet destined to port 4. In the absence of failures, this packet will enter the switch with `status` = 1111 and get assigned `tag` = 4 in $T_1$. It will then match the $4^{\text{th}}$ entry in the second table $T_2$ and be forwarded on port 4. When port 4 fails (*i.e.*, it is not active), the same packet will now match the $5^{\text{th}}$ entry in $T_2$. This will modify `tag` to 1 and the packet will be recirculated, now matching the $1^{\text{st}}$ entry and being routed on port 1.

**Packet recirculation degrades flow completion time.** There are few drawbacks with the above implementation: when a packet is recirculated, *i)* it creates a "self-induced incast" on the ingress buffer, consuming extra bandwidth, *ii)* it increases the packet processing latency since the same packet needs to go through the match-action pipeline (including its buffers) multiple times. To understand the impact of recirculating packets, we ran a series of simulations using the ns3 discrete-event simulator. We validated our ns3 model with a manufacturer of hardware PDPs. We took an existing ns3 implementation from the state-of-the-art datacenter load-balancing codebase (*i.e.*, Hermes [22]) and implemented the F10 [9] state-of-the-art FRR mechanism. The network topology is a 2-tier leaf-spine datacenter topology, the congestion control is DCTCP, and the routing is OSPF/ECMP. Refer to §V for detailes about the datacenter setting. In Fig. 3, we failed one or two links simultaneously and compared an "ideal" OSPF routing approach that reconverges at the time of the failure (*i.e.*, "CP reconvergence") with the packet recirculation approach (*i.e.*, "FRR recirculation"). Our results show that the flow completion time (FCT) of latency-sensitive flows (*i.e.*, small flows with size $\leq$ 100 KB) is a factor of 2.4x and 3.7x higher with "FRR recirculation" under one and two link failures, respectively, compared to CP-reconvergence. We also measured the average throughput achieved by the large flows (*i.e.*, size $\geq$ 10 MB) when recirculating packets, which achieved a 2.7x and 3.3x times lower throughput than CP-reconvergence under one and two link failures, respectively.

**A sequential search of the first active port wastes hardware resources.** Another way to implement the above FRR on a match-action pipeline would be to either sequentially or simultaneously check through a specific sequence of outgoing ports, which port is the first active one. This approach can easily be expressed in P4 as a set of nested "if-else" statements and the compiler has to decide whether to realize it in a sequential (on SRAM memory) or parallel (on TCAM memory) manner. In the sequential case, the status of each port in an FRR sequence is tested in each subsequent stage of the match-action pipeline. This approach has two clear limitations: *i)* it cannot support FRR sequences whose sizes are larger than the number of stages and *ii)* it wastes resource at each stage that cannot be used by forwarding functions that have a functional dependency with the selected egress port.

**A TCAM-based parallel search to the rescue!** A P4 compiler can encode a set of if-else statements within a TCAM memory, anabling a parallel active-port search. We present a naïve encoding approach in Fig. 4a where we realize the same circular FRR sequences of the packet recirculation case with one single TCAM lookup. We assign an identifier $FRR_{id}$ to each FRR sequence. When a packet arrives at the switch, we attach both the *status* metadata field and a given $FRR_{id}$ to it. We then match the packet with the TCAM memory and extract the first active forwarding port in one single TCAM lookup. As an example, the first four entries in the table realize the FRR sequence $\langle 1,2,3,4\rangle$. We now compute the amount of TCAM space needed to realize a *set* of $n$ circular FRR sequences using the aforementioned naïve TCAM encoding. If the number of ports in each sequence is $k$, then the number of TCAM entries will be $nk$ and the TCAM occupancy is $nk(k+\log n)$, where we need $\log n$ bits to encode FRR identifiers and $k$ bits to encode the `status` match part for each of the $nk$ entries. In the specific example of Fig. 4a, we can see that just a single circular FRR sequence requires 4 TCAM entries and thus 24 bits of TCAM memory. Observe that already for $k = 24$ and 10 sets of FRR circular sequences (each set has 24 sequences — all cyclic shift options), we need 5760 TCAM entries and $\sim$ 130 kbit of TCAM space, which is already two orders of magnitude larger than what is available in today's high-performance PDPs [12]. In the remaining sections, we therefore address the following main question:

*"Can we enable a new FRR primitive for programmable data planes that requires minimal TCAM overhead while minimizing flow performance degradation due to network failures?"*

| Table $T_1$ | | |
|---|---|---|
| $\text{FRR}_{id}$ | status | fwd |
| 1 | 1*** | 1 |
| 1 | *1** | 2 |
| 1 | **1* | 3 |
| 1 | ***1 | 4 |
| 2 | *1** | 2 |
| 2 | **1* | 3 |
| 2 | ***1 | 4 |
| 2 | 1*** | 1 |
| 3 | **1* | 3 |
| 3 | ***1 | 4 |
| 3 | 1*** | 1 |
| 3 | *1** | 2 |
| 4 | ***1 | 4 |
| 4 | 1*** | 1 |
| 4 | *1** | 2 |
| 4 | **1* | 3 |

(a) Naïve approach

| Table $T_1$ | |
|---|---|
| $\text{FRR}_{id}$ | port_set |
| 1 | 1111000 |
| 2 | 0111100 |
| 3 | 0011110 |
| 4 | 0001111 |

$\downarrow$

| Table $T_2$ | | |
|---|---|---|
| port_set | status | fwd |
| 1****** | 1*** | 1 |
| *1***** | *1** | 2 |
| **1**** | **1* | 3 |
| ***1*** | ***1 | 4 |
| ****1** | 1*** | 1 |
| *****1* | *1** | 2 |
| ******1 | **1* | 3 |

(b) Encoded approach

Figure 4: TCAM encodings of a circular FRR sequence.

## III. A Primitive for Fast Reroute

Here, we provide an approach for *encoding* an arbitrary set of FRR sequences into a match-action TCAM-based packet processing pipeline. We discuss how to do that when the sequences are "circular", as in a wide variety of FRR mechanisms that have been proposed [9], [23], [24], [**?**]. Then, we devise three different heuristics that efficiently encode any type of arbitrary FRR sequences into TCAM memories.

### *A. A Model for Programmable FRR*

**Fast ReRoute (FRR) sequences.** Network operators rely on FRR mechanisms to compute a set of primary and backup forwarding rules. These rules are used to reroute network traffic upon arbitrary number of failures without the need to invoke the slower control plane. When a switch receives a packet, it classifies it, possibly modifies the packet header, and finally applies a forwarding action. In this paper, we model each forwarding action with an *FRR sequence*, *i.e.*, a sequence of ports, *e.g.*, $\langle port1, port4, port2, port3\rangle$, or $\langle 1, 4, 2, 3\rangle$ for brevity. A switch forwards packets to the first (traversing from left to right) active port in a sequence. For instance, when all ports are active, a switch using the FRR sequence $F_0 = \langle 1, 2, 3, 4\rangle$ will forward packets through port 1. If both ports 1 and 2 fail, the switch reroute packets through port 3. Packets belonging to different flows may share the same forwarding behavior, that is, the same FRR sequence.

**Target-dependent constraints.** The architecture of a packet processing system highly influences the way FRR sequences would be supported. For instance, a software switch cannot leverage dedicated memories for ternary matching (*i.e.*, TCAMs). Even among switches with TCAM support there are differences to be taken into account. As an example, Intel FlexPipe [25] does not support arbitrary width sizes for TCAM tables, a functionality that is supported in the RMT (Reconfigurable Match Tables) architecture [26]. We note that these details are not exposed to the P4 programmer but handled by target-dependent P4 compilers. In this paper, we focus our attention on the emerging PDPs that support wildcard match tables (*e.g.*, TCAM memories). We now describe a set of architectural constraints for hardware PDPs.

- *Match-action pipeline stages.* There are a fixed number of stages through which packets are being classified and modified. Some stages may allow to perform parallel matches in different tables (*e.g.*, FlexPipe) and each stage contains a certain amount of resources for exact, prefix, and ternary matches. As noted in §II, implementing FRR sequences in a sequential manner is highly undesirable in practice. In fact, it prevents any forwarding operations with a functional dependency on the egress port calculation to leverage the spare SRAM and TCAM memories that reside within the stages used to implement the FRR sequences. We therefore require the bulk of our encoding to fit within a single stage (a small table can be allowed in the previous stage to assign FRR identifiers and initialize data structures).
- *Number of TCAM entries and bits.* Each stage $s$ of the match-action pipeline has a certain number of TCAM entries. For instance, the RMT architecture states a maximum of 32K TCAM entries per stage, though this amount may be smaller in practice depending on the specific vendor and product [12].[1] In FlexPipe, there are only two stages with 12K entries each. In each stage $s$, the amount of TCAM memory (in bits[2]) is also limited. In the RMT architecture, roughly 1 Mbit of TCAM memory is available per stage.

**FRR encoding goal.** Our objective is to provide a primitive that allows efficient realization of any set of FRR sequences. We already explained in §II that such a solution must be based on a single TCAM lookup implementation. Given a set of FRR sequences that correspond to a specific fast failover algorithm (*e.g.*, DFS traversal [24] or circular-arborescence [27]) our proposed primitive will allow deploying them in a way that reduces the amount of TCAM memory required.

### *B. A Primitive for Circular FRR*

We now describe a TCAM scheme for encoding a specific class of widely adopted FRR sequences, *i.e.*, circular FRR sequences. This class of FRR sequences is common of several existing FRR mechanisms, including F10 [9], arc-disjoint arborescences [27], and graph-traversals [24]. We say that a set of FRR sequences is *circular* if every FRR sequence in the set can be obtained from any other sequence by a finite number of circular shift operations. Consider a switch with four ports and the following set of FRR sequences: $F_1 = \langle 1, 2, 3, 4\rangle$, $F_2 = \langle 2, 3, 4, 1\rangle$, $F_3 = \langle 3, 4, 1, 2\rangle$, and $F_4 = \langle 4, 1, 2, 3\rangle$. Since every $F_i$ can be obtained from any other $F_j$ by circularly shifting $F_i$ to the left $j - i \bmod 4$ times, the FRR sequences in the set $\{F_1, F_2, F_3, F_4\}$ are circular.

**Encoding circular FRR sequences.** We already described a naïve approach for encoding circular FRR sequences in §II, which was illustrated in Fig. 4a. As discussed earlier, this approach requires $nk(k + \log n)$ TCAM bits, where $n$ is the number of sets of circular FRR sequences and $k$ is the number of ports of the switch (and hence, the length of an FRR sequence). Let us now propose a more efficient way of

[1] Also based on private communication with vendors.

[2] For simplicity, we use the "bit" terminology as opposed to the more correct "trits" one, which captures the ternary nature of the TCAM elements.

encoding any set of circular FRR sequences (see Fig. 4b). Let $f_{i,j}$ represent the $j$'th element of a sequence $F_i$. For each sequence $F_i$, we assign a bit vector port_set of size $2k-1$, where each bit represents a port of the switch in the order defined by the sequence $F_1$, *i.e.*, bit number $b$ of port_set represents port $f_{1,b \bmod k}$. For each sequence $F_i$ we set $k$ bits in its port_set vector that correspond to the ports in $F_i$ but in the same order that the ports appear in $F_i$. In our example (Fig. 4b), the port_set vector represents ports $\langle 1,2,3,4,1,2,3\rangle$. Hence, for the sequence $F_1$, the port_set is 1111000, which means that the bits corresponding to ports $\langle 1,2,3,4\rangle$ are set to 1. For the sequence $F_3$, we will have port_set $=$ 0011110 which means that the bits corresponding to ports $\langle 3,4,1,2\rangle$ are set.

Table $T_1$ in Fig. 4b assigns the corresponding port_set for each circular sequence of a given FRR set. Then, table $T_2$ matches the port_set and the status metadata fields to determine the first active port for a given FRR sequence. For example, if a packet needs to be rerouted according to sequence $F_4$ (this is determined at an earlier stage, not shown here), then table $T_1$ will assign it port_set $=$ 0001111. Now, let's assume that ports 1 and 4 are not active and ports 2 and 3 are active, which corresponds to the status $=$ 0110. Then, the first matching entry in table $T_2$ will be in row 6 (where port_set $= *****1*$) and thus, the packet will be forwarded via port 2. Notice that different circular FRR sets will be assigned different $FRR_{id}$ in table $T_1$, and thus will have dedicated sets of entries in table $T_2$.

**Our encoding achieves an order of magnitude smaller TCAM memory usage compared to a naïve approach.** Let us analyze the TCAM space required to encode a set of $n$ circular FRR sequences, each of length $k$ (notice that there are at most $k$ such sequences, *i.e.*, $n \le k$). The table $T_1$ requires $n$ entries, each of size $\log n$ bits. The table $T_2$ requires $2k-1$ entries, each of size $2k-1+k$ bits. So, the total TCAM space required for a single FRR set is $n\log n + (2k-1)\times(3k-1) = O(k^2)$. This is an order of magnitude better than the naïve approach which requires $nk(k+\log n) = O(k^3)$ TCAM bits. Moreover, table $T_1$ does not need ternary matches, thus can be then implemented in SRAM, further saving expensive TCAM space.

## C. A Primitive to Implement Them All

We now introduce the general problem of *encoding* an arbitrary set of FRR sequences that are not necessarily circular. The input is a set of sequences and the output is the set of wildcard (TCAM) and exact (SRAM) matches and actions to be installed in the forwarding plane. The aim is to generalize the port_set vector described in the previous subsection.

**Single-table optimization.** We first consider the problem of encoding a set of FRR sequences in a single TCAM table. The challenge with arbitrary FRR sequences is that the mapping between bits in the port_set vector and ports is not as obvious as it was in the circular case. The port_set now has to represent a sequence of ports that contains all the given FRR sequences as subsequences. Essentially, this means finding the shortest sequence that contains all the given sequences as subsequences (*i.e.*, skipping elements is allowed).

**Algorithm 1** Definition of GREEDY.

```
Global parameters: A constant $d \in \mathbb{N}$
Input: A set $\mathcal{F} = \{F_1,\dots,F_{cd}\}$ of FRR sequences
1) Set $currscs := \langle\rangle$
2) Repeat for each $i = 1,\dots,c$
     ◦ $currscs :=$ DP-SCS ($currscs, F_{(i-1)d+1},\dots,F_{id}$)
3) return $currscs$
```

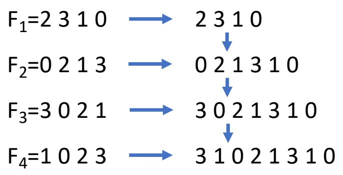


Figure 5: GREEDY example.

**Unveiling an unexplored connection between FRR encodings and algorithmic string theory.** Our encoding problem can be seen as a special (and unexplored) version of the classic Shortest Common Supersequence (SCS) [28] problem, *where no repetitions are allowed.* In the SCS problem, the input is a set of sequences $\mathcal{S} = \{S_1,\dots,S_k\}$ and the goal is to compute a sequence of elements $\bar{S}$ such that any element of $\mathcal{S}$ is a subsequence of $\bar{S}$ and $\bar{S}$ is of minimal size. This connection is interesting and raises the question whether our version of the problem without repetitions can render the problem simpler: SCS is known to be notoriously hard, in fact NP-hard already for strings over a binary alphabet [29], and also hard to approximate within polylogarithmic factors [30]. Unfortunately, this is not the case: we state this insight as a theorem as the result is of independent interest.

**Theorem 1.** *The SCS problem without repetitions is $NP$-hard to optimize and approximate.*[3]

The proof follows directly from [30].

**The dynamic programming building block: DPSCS.** We first discuss a well-known technique used to solve the SCS problem optimally based on Dynamic-Programming [31], called DPSCS. This approach computes an optimum SCS solution in time $O(k^n)$, thus solving the problem in efficient (polynomial) time only when the number of sequences is constant. We use DPSCS as a baseline to compare our heuristics and to deal with arbitrary number of sequences. The input to our problem is a set $\mathcal{F} = \{F_1,\dots,F_n\}$ of FRR sequences, where $f_{i,j}$ indicates the $j$'th element of sequence $F_i$. The value of $f_{i,j}$ represents an index of a port in the switch. We assume that all the sequences have the same length $k$.

**The GREEDY heuristic.** We present a novel greedy algorithm GREEDY (Alg. 1) which is based on iteratively applying the optimal DPSCS approach to a subset of sequences. GREEDY first partitions the FRR sequences $\mathcal{F}$ into small groups of con-

[3]There exists a constant $\delta > 0$ such that, if SCS has a polynomial-time approximation algorithm with ratio $\log^{\delta} n$, where $n$ is the number of input sequences, then NP is contained in DTIME($2^{\mathrm{polylog}\, n}$).

| table $T_1$ | |
|---|---|
| FRR$_{id}$ | port_set |
| 1 | 00010111 |
| 2 | 00111100 |
| 3 | 10111000 |
| 4 | 01110100 |

$\rightarrow$

| table $T_2$ | | |
|---|---|---|
| port_set | status | fwd |
| 1******* | ***1 | 3 |
| *1****** | *1** | 1 |
| **1***** | 1*** | 0 |
| ***1**** | **1* | 2 |
| ****1*** | *1** | 1 |
| *****1** | ***1 | 3 |
| ******1* | *1** | 1 |
| *******1 | 1*** | 0 |

Figure 6: GREEDY TCAM implementation.

stant size $d$, which are then solved optimally using DPSCS. More specifically, GREEDY merges subsolutions sequentially, by feeding the DPSCS subroutine with ($currscs$, $F_{(i-1)d+1}$, $\ldots$, $F_{id}$) where $currscs$ is the intermediate SCS solution. We show an example of GREEDY in Fig. 5 with four sequences $F_1$=$\langle 2\ 3\ 1\ 0\rangle$, $F_2$ =$\langle 0\ 2\ 1\ 3\rangle$, $F_3$=$\langle 3\ 0\ 2\ 1\rangle$, and $F_4$=$\langle 1\ 0\ 2\ 3\rangle$, where $d = 1$. GREEDY first computes the SCS between $F_1$ and $F_2$ using DPSCS, obtaining the sequence $\langle 0\ 2\ 1\ 3\ 1\ 0\rangle$. It then computes the SCS between this sequence and the next one, that is, $F_3$, again using DPSCS on just these two sequences. The output is then fed as input to the last SCS computation with $F_4$, returning a sequence of 8 elements. We show the TCAM implementation of this sequence in Fig. 6. The dynamic program DPSCS is based on a $(n \times n+1)$-dimensional matrix $M$ and can be used to compute the shortest common supersequence when the number of input sequences, $n$, is constant. The complexity of GREEDY is $\frac{n}{d}O(k^d)$, and assuming $d$ is constant, we have $O(nk^d)$.

**The HIERARCHICAL heuristic.** An alternative approach is to merge subsequences *hierarchically* (in a tournament fashion), rather than sequentially like in GREEDY.[4] This idea is pursued by the HIERARCHICAL algorithm (Alg. 2). As we will see, such an algorithm is faster than GREEDY and computes subsequences of length similar to GREEDY. Like GREEDY, HIERARCHICAL uses DP-SCS to compute optimal solutions in polynomial time for a constant number of sequences, splitting $\mathcal{F}$ into $d$ sets $\mathcal{M}_1, \ldots, \mathcal{M}_d$. However, unlike GREEDY, HIERARCHICAL merges these optimal sequence *hierarchically*, using DP-SCS (H-SCS($\mathcal{M}_1$), $\ldots$, H-SCS($\mathcal{M}_d$)). In Fig. 7, we show an example with HIERARCHICAL using the same four sequences as in the GREEDY example and setting $d = 2$. The lowest level of the recursion in HIERARCHICAL computes the SCS among pairs of sequences using DPSCS, *i.e.*, $F_1$ with $F_2$ and $F_3$ with $F_4$. The two resulting SCS sequences are fed as input to a final SCS computation (again using DPSCS) in order to obtain the output of HIERARCHICAL. The asymptotical complexity of this algorithm is the same as GREEDY. At the lowest level of the hierarchy, there will be $\frac{n}{d}$ executions of the DP-SCS; at the previous level we have $\frac{n}{d^2}$ executions, and so on. This results in $O(nk^d)$ complexity.

**The FAST-GREEDY heuristic.** The DPSCS algorithm computes optimal solutions at the cost of running time, *i.e.*, exponential time in the number of sequences. For this reason, we introduce FAST-GREEDY (Alg. 3), which strikes a different trade-off in terms of fast running time and reasonably good accuracy. At each iteration, we trim the left-most element

[4]We note that this algorithm has traditionally been used to solve general SCS problems [32], thus we use it only as a means of comparison.

**Algorithm 2** Definition of HIERARCHICAL (H-SCS).

```
Global parameters: A constant $d \in \mathbb{N}$
Input: A set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of FRR sequences
  1) If $|\mathcal{F}| \leq d$
       ◦ return DP-SCS ($\mathcal{F}$)
  2) else
       a) split $\mathcal{F}$ into $d$ sets $\mathcal{M}_1, \ldots, \mathcal{M}_d$
       b) return DP-SCS (H-SCS($\mathcal{M}_1$), ..., H-SCS($\mathcal{M}_d$))
```

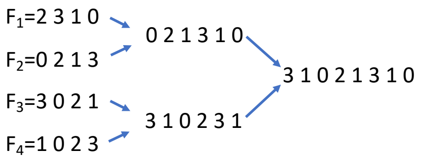


Figure 7: HIERARCHICAL example.

**Algorithm 3** Definition of FAST-GREEDY.

```
Input: A set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of FRR sequences
each of length $k$, where $f_{i,j}$ is the $j$'th element of
sequence $F_i$.
  1) Set $currscs := \langle\rangle$
  2) Repeat until $\exists i \in [1, \ldots, n], |F_i| > 0$
       • Let $\mathcal{S} = \{i \mid |F_i| = m, i \in [1, \ldots, n]\}$, where
         $m = \max_i |F_i|$
       • Let $a$ be the most frequent element in $\{f_{i,1} \mid i \in \mathcal{S}\}$
       • $\forall i \in \mathcal{S}$, if $f_{i,1} = a$ then $F_i = \langle f_{i,2}, \ldots, f_{i,k}\rangle$
       • $currscs := currscs \cup \langle a\rangle$
  3) return $currscs$
```

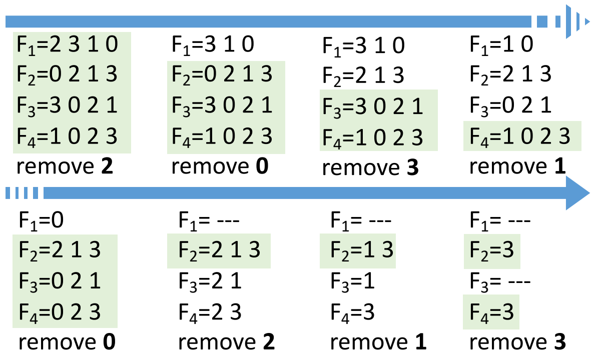


Figure 8: FAST-GREEDY example.

from some of the input sequences according to the following approach. First, the algorithm identifies the set $\mathcal{S}$ of the longest sequences at the current iteration. Then, it looks at the leftmost elements of all these longest sequences and identifies the one that appears most often (ties are broken arbitrarily). This "most-frequent" element (denoted as $a$) is removed from the sequences in $\mathcal{S}$ where it appears as the left-most element, and added to the resulting SCS sequence. The process continues until all the input sequences are empty. The running time of FAST-GREEDY is $O(n^2k)$ — much faster than any $O(k^n)$ DPSCS-based heuristic, where $k$ is the size of a

| Table $T_1$ | |
|---|---|
| FRR$_{id}$ | port_set |
| 1 | 10111000 |
| 2 | 01010101 |
| 3 | 00101110 |
| 4 | 00011101 |

$\rightarrow$

| Table $T_2$ | | |
|---|---|---|
| port_set | status | fwd |
| 1⋆⋆⋆⋆⋆⋆⋆ | ⋆⋆1⋆ | 2 |
| ⋆1⋆⋆⋆⋆⋆⋆ | 1⋆⋆⋆ | 0 |
| ⋆⋆1⋆⋆⋆⋆⋆ | ⋆⋆⋆1 | 3 |
| ⋆⋆⋆1⋆⋆⋆⋆ | ⋆1⋆⋆ | 1 |
| ⋆⋆⋆⋆1⋆⋆⋆ | 1⋆⋆⋆ | 0 |
| ⋆⋆⋆⋆⋆1⋆⋆ | ⋆⋆1⋆ | 2 |
| ⋆⋆⋆⋆⋆⋆1⋆ | 1⋆⋆⋆ | 1 |
| ⋆⋆⋆⋆⋆⋆⋆1 | ⋆⋆⋆1 | 3 |

Figure 9: FAST-GREEDY TCAM implementation.

FRR sequence. Note that we look at the most frequent element among the longest sequences as this helps in making progress over all the sequences. In Fig. 8, we show an example of FAST-GREEDY with four sequences $F_1$=$\langle 2\ 3\ 1\ 0\rangle$, $F_2$ =$\langle 0\ 2\ 1\ 3\rangle$, $F_3$=$\langle 3\ 0\ 2\ 1\rangle$, and $F_4$=$\langle 1\ 0\ 2\ 3\rangle$. We highlight with a green background the longest sequences during the computation, which are those sequences from which we extract the most frequent element. At the beginning, all sequences have the same length and all the left-most elements appear exactly once. The algorithm selects 2 as the most frequent element and removes it from all the sequences where it appears as the left-most element, *i.e.*, only from $F_1$. FAST-GREEDY then applies the same procedure until the input sequences are empty. Consider the 3rd stage where FAST-GREEDY selects element 3 as the most frequent and removes it. The element is removed from $F_3$ (where we selected it) and also from $F_1$ where it appears as the left-most element. The final supersequence is $\langle 2\ 0\ 3\ 1\ 0\ 2\ 1\ 3\rangle$. By iteratively removing the common left-most elements of each subsequence, we can guarantee the final sequence will be a supersequence of each individual subsequence.

We now analyze the computational complexity of FAST-GREEDY. At each iteration, finding the most frequent left-most element costs $O(n)$ and each element is removed exactly once so the number of removals is $O(nk)$. Thus, the running time of this algorithm is $O(n^2k)$.

**Multi-table optimization.** Here we consider the problem where the FRR encoding can be realized across multiple tables instantiated in the same pipeline stage, which is possible on today's programmable switches [33]. This allows to build even more compact representations of a set of FRR sequences. In some cases, using multiple tables may also be necessary as hardware switches cannot handle tables of arbitrary width, *e.g.*, 512 bits. We describe a heuristic that carefully groups FRR sequences based on a *novel* insight into the algorithmic theory of strings, which is tailored for the specific case of FRR sequences (*i.e.*, no element repetitions). As an example, consider the same FRR sequences used for the previous heuristics (see Fig. 8). Initially, $S = S' = \{(2,3,1,0),(0,2,1,3),(3,0,2,1),(1,0,2,3)\}$ and assume the maximum TCAM width is 10 bits. Clearly, we cannot realize the solution obtained from FAST-GREEDY since it requires 8 TCAM bits for the port_set and 4 bits for the status. We can however create two tables, each of 10 bits for the TCAM width. In the first table, the enconding of the port_set is $(0,2,3,1,0,3)$ while the encoding in the second table is $(1,3,0,2,1,3)$. This requires 10 bits per table and encodes the first (last) two FRR sequences in the first (second) table.

**Algorithm 4** Definition of MULTITABLE-SCS (MT-SCS).

```
Function input: A set $\mathcal{F} = \{F_1, \ldots, F_k\}$ of FRR sequences, and a max TCAM width of $t > 0$.
1) Let $S = \{\}$, add $\{F_1\}, \ldots, \{F_k\}$ into $S$, and let $f$ = True
2) Repeat until $f$ is True or $\exists f \in S$ s.t. $|f| > t$
    a) $S' = S$ and $(S_i, S_j) := \max_{i,j} LCS(S_i \cup S_j)$
    b) add $\{S_i, S_j\}$ into $S'$ and remove $S_i$ and $S_j$ from $S'$
    c) if $cost(S) \leq cost(S')$ and $\nexists f \in S$ s.t. $|f| > t$, then
       $f$ =False; else $S = S'$
3) return $S$
```

**Algorithm 5** FAST-LCS: LCS without repetitions.

```
Function input: A universe $U$ of elements and a set
$\mathcal{F} = \{F_1, \ldots, F_n\}$ of sequences each of length $r$, where
$f_{i,j} \in U$ indicates the $j$'th element of sequence $F_i$.
1) Build $G(V, E)$ where $V = U \cup \{s\}$ contains a node for
   each element in $U$ and $E$ contains a directed arc $(a, b)$
   if $a$ appears before $b$ in all sequences. $E$ also contains
   $(s, v)$ for all $v \in U$.
2) Compute the longest paths from $s$ to any vertex of $G$
   through a topological sorting of $G$ from $s$.
3) return the longest path
```

**The MULTITABLE-SCS heuristic.** One way to "pack" FRR sequences into multiple tables is to aggregate similar FRR sequences together. Intuitively, this allows similar sequences to share a small port_set vector, potentially achieving lower memory overhead than with a single table. Finding similar sequences leads us to consider a complementary problem to SCS, *i.e.*, the Longest Common Subsequence (LCS) [34] problem.[5] LCS is renowned for being NP-hard, but again, in our context, we do need to consider LCS *with a tweak*: we do not have any repetitions. This again poses the problem of whether the NP-hardness of the LCS holds without repetitions. Interestingly, in this case, we find that this version can be solved *efficiently*, in polynomial time (*i.e.*, $O\left(nk^2\right)$). This result is of independent interest.

**Theorem 2.** *The LCS problem without repetitions is polynomial-time solvable.*

*Proof.* The proof is constructive and based on Alg. 5. The first algorithm is by reduction: we note (step (1)) that we can build a directed graph between the characters as follows: there is an arc from character $a$ to character $b$ if $a$ is before $b$ in every string. Now observe that only characters connected with arcs can appear in the LCS at the same time. The graph must be acyclic by construction. The problem boils down to finding the longest path in an acyclic directed graph, which can be solved efficiently where $k$ is the size of a sequence. $\square$

We consider LCS as a way to efficiently group FRR sequences into different tables so that the encoding of each group of sequences fits within the maximum TCAM width $t$ or it produces an overall smaller cost than having a single

[5]Note that, formally, LCS is not the dual problem of SCS.

table. In MULTITABLE-SCS (Alg. 4), we divide the input FRR sequences into $n$ sets (step (1)) and then aggregate the two sets $S_i$ and $S_j$ with the largest LCS (steps (2a) and (2b)). If aggregating these elements produces a lower memory cost or reduces the amount of violations of the TCAM maximum width, we repeat the procedure. We stop it otherwise and return the set partitioning, each set corresponding to a table encoding.

## IV. IMPLEMENTATION

In order to verify the feasibility of our primitive, we made several implementations. In the following, we will first report on P4-based implementations (*i.e.*, bmv2 [13] and Tofino) and will then discuss a Verilog implementation on the NetFPGA.

**P4-based implementations.** We successfully implemented our primitive for a number of existing FRR mechanisms, including arborescence-based FRR mechanisms [23], as well as the Depth First Search (DFS), Breadth First Search (BFS) and the rotor router mechanisms in [35]. We also successfully implemented our primitive on the Tofino switch, further confirming the feasibility of our approach. We will share our implementations together with this paper. We note that implementing PURR in P4 simply requires to install the two tables showed in Fig. 4b in the existing forwarding pipeline. The first table only requires an exact match operation while the second table requires the most complex wildcard match.

**FPGA-based implementation.** We built our prototype on the NetFPGA-SUME [15], which is a PCIe adapter card with 4x10 Gbps Ethernet interfaces and an FPGA Xilinx Virtex-7.

We leveraged the existing layer-2 switch implementation provided with NetFGPA-SUME package to deploy PURR. In this system, packets first enter the device through one of the four 10 Gbps network interfaces where packets are stored in First-In-First-Out (FIFO) memory units, named input queues. The interface modules are connected to the input arbiter. The arbiter switches between the input queues in a round robin fashion, each time selecting a non-empty queue and moving one packet from it to the next stage in the data path. From the input arbiter on, there is a single pipeline with a data width of 256 bits running at the frequency of 200 MHz, thus guaranteeing enough bandwidth to support 40 Gbps transmission rates. The forwarding logic comes after the input arbiter. It is responsible for selecting the output port based on standard layer-2 switching operation. After the decision is made, the packet reaches the PURR primitive logic. Here, constant monitoring of the physical network interfaces status is needed to activate the programmed FRR mechanism. Indeed, the appropriate output port is selected based on the status of the physical network interfaces and the result of a matching against the TCAM memory. If the originally selected destination port is active, then nothing changes. In contrast, if the selected port is down, the new destination port will be selected based on the TCAM matching result, which depends on the adopted FRR algorithm.

## V. EVALUATION

We now assess the performance of the algorithms introduced in §III for encoding a set of FRR sequences into a TCAM memory. We evaluate them along two dimensions: the amount of memory (bits) needed to encode the FRR sequences and their running time. First we focus on a single switch that gets a set of FRR sequences as input and encodes them in a TCAM memory. Then, we set up a datacenter Clos network and implement the state-of-the-art FRR mechanism for Clos networks, *i.e.*, F10 [9], using circular FRR sequences. Using this scenario, we run simulations in ns3 to study the impact of using PURR w.r.t. an approach based on recirculating packets.

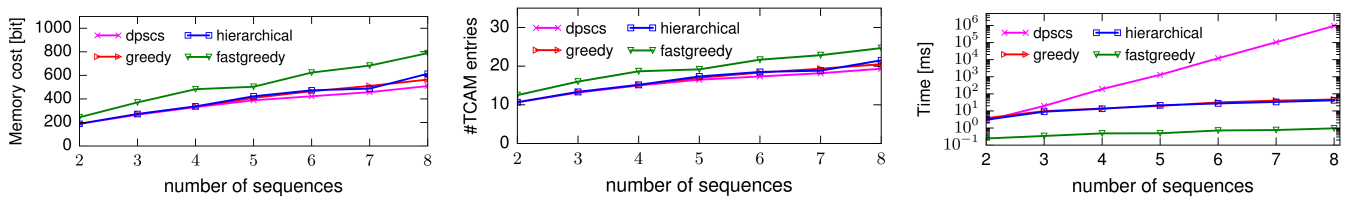
### A. FRR Encoding

In this section, we answer the following question: "*How much TCAM memory (in bits and entries) do we need to implement a given set of FRR sequences?*". We implement DPSCS, GREEDY, HIERARCHICAL, and FAST-GREEDY in Python and consider three different dimensions: *i)* the number of FRR sequences $n$, *ii)* the size $k$ of the FRR sequences, *iii)* we either generate random sequences or construct sequences derived from existing FRR mechanisms. For each simulation setting, we run 6 simulations with different seeds.

**Encoding FRR sequences is crucial in high port density switches.** We first evaluate the NAÏVE approach described in Fig. 4a and compare with our encoding-based mechanism described in Fig. 4b. The results are based on the calculations described in §III-B. We consider the family of FRR mechanisms (*e.g.*, F10 [9], DFS [24], basic arc-disjoint spanning trees [23], [**?**]), which rely on circular FRR sequences. Realizing a circular FRR sequence over 8, 16, 32, and 64 ports takes 1.5x, 2.8x, 5.5x, and 10.8x higher memory requirements than using an encoding-based implementation, respectively. A PDP with 64 ports would require 327 KB of TCAM to implement 10 circular FRR sequences. This corresponds to 2 pipeline stages on the RMT architecture and 5 stages in other programmable data planes [12]. An encoded approach would require 30 KB, one tenth of the TCAM memory contained in a single stage of the RMT architecture [26].

**FAST-GREEDY performs close to the optimum and is fast.** We now compare FAST-GREEDY against the optimum SCS solver, *i.e.*, DPSCS. We set the size of the sequences to 7 elements and vary the number of sequences from 2 to 7. Fig. 10a and Fig. 10b show that FAST-GREEDY performs remarkably close to the optimum while it consumes roughly 20% more TCAM bits and 10% more TCAM entries than the optimum. We report the processing time in Fig. 10c. As expected, dynamic programming grows exponentially in the number of sequences, requiring 15 minutes to find the optimum SCS for even just 8 sequences. In contrast, FAST-GREEDY runs in less than one millisecond.

**FAST-GREEDY works best on large sets of sequences.** We compare our three heuristics using larger instances. We plot our results for the consumption of TCAM memory in Fig. 11a, Fig. 11b, and Fig. 11c for sequences with 8, 16, and 32 elements each, respectively, followed by their corresponding running times in Fig. 11d, Fig. 11e, and Fig. 11f, respectively. We draw two main conclusions. First, for large number of sequences (*i.e,* $\geq$ 100), FAST-GREEDY outperforms both



(a) Memory consumption in TCAM bits. (b) Memory consumption in TCAM entries. (c) Processing time.

Figure 10: Comparison of FAST-GREEDY with respect to the optimum. The size of the sequences is set to 7.

GREEDY and HIERARCHICAL in both TCAM memory utilization and running times. Second, GREEDY and HIERARCHICAL require one second to process merely 20 sequences. FAST-GREEDY can process tens of thousand of sequences in the same amount of time, thus achieving higher scalability.

**FAST-GREEDY compresses hundreds of thousands of FRR sequences within limited memory.** Fig. 12a and Fig. 12b show the amount of memory in bits and the number of entries required to implement a given set of FRR sequences. By doubling the number of ports on a switch, the number of TCAM entries increases roughly by a factor of 3.5x while the number of TCAM bits increases by a factor of 7x. The required memory stabilizes around 1000 FRR sequences, after which the encoding is capable of realizing the vast majority of possible FRR sequences provided as input to FAST-GREEDY.

**Compressing short sequences provides larger memory saving.** We also evaluated the case where we have a switch with a large number of ports but the length of the sequences is small. For instance, on a 64-port switch, an operator may define FRR sequences of size 5 to protect against *any* arbitrary 4 possible link failures. Using the *Naïve* approach described in Fig. 4a, one would have to define all possible 4 elements sequences out of 64 elements, *i.e.*, $64!/(64-4)!$ 900 million sequences each requiring 5 TCAM entries. Using PURR, one can compute a single SCS containing the 64 ports repeated five times, *i.e.*, 320 TCAM entries. We refer to the ratio between the memory (in bits) used by the *Naïve* approaches and the memory used by PURR as the memory savings. In Fig. 12c, we show the memory savings (y-axis) with PURR for increasing sizes of FRR sequences (x-axis) and different number of ports on the switch (different lines) in percentages. We note that on a switch with 8 ports (green line) and FRR sequences of size 8, PURR uses $\sim 0,1\%$ of the memory used by the *Naïve* approach. When the switch has 16, 32, or 64, PURR reduces the memory requirements by 7, 9, 11 orders of magnitudes (not visible in the figure), respectively. We can observe that the memory savings exist also for very short sequences of just two elements per FRR sequence and grows exponentially for increasing sizes of FRR sequences.

**Memory requirements of state-of-the-art FRR mechanisms.** We so far evaluated the memory requirements when the input of the problem consisted of randomly derived FRR sequences. One may ask whether existing FRR mechanisms (robust to multiple failures) would require higher or lower memory than random sequences. To the best of our knowledge, the best general FRR mechanisms that are *i)* scalable, *ii)* robust to *multiple* failures, and *iii)* do not require expensive transactional high-speed memories on the chip are those based on computing a set of "arc-disjoint" spanning trees [23], [**?**]. We quantify the memory requirements of an arc-disjoint FRR mechanism, called *tree*, in Fig. 13 deployed on Jellyfish [36] datacenter topologies. Through *tree*, all the spanning trees are ordered in a sequence and a packet is rerouted once on the next spanning tree and once "bounced" on the opposite tree each time it hits a failed link. Our results show that the FRR sequences created via tree-based FRR approaches induce the same memory requirements of random sequences.

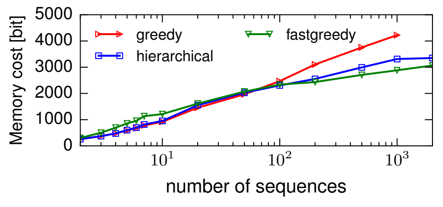
**Multiple tables.** We ran simulations using random sequences in order to assess the benefits of splitting a set of FRR sequences into multiple tables. In each simulation, we generate between 10 and 100K different random FRR sequences and run the LCS-based MULTITABLE-SCS algorithm where the *cost* function minimizes the amount of TCAM bits. We observe that the algorithm always returned a single table, thus showing limited benefits in splitting a table into multiple tables (unless some TCAM width constraints apply). We note that all our encodings would fit in the TCAM width of the RMT pipeline architecture in one single stage [26].

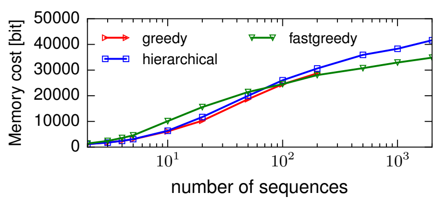
## B. Datacenter Simulations

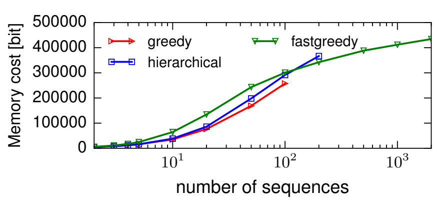
We now investigate the following main question: *How does the FCT of latency-sensitive flows and the throughput of bandwidth-intensive applications vary depending on the implemented FRR primitive?* We assess the impact of our FRR primitive on a real datacenter workload. We note that PURR can be also applied to other types of networks, *e.g.,* WANs. We compare PURR against the performance achieved using *i)* an FRR primitive based on recirculation ("recirc"), *ii)* an *ideal* immediate reconvergence of the control-plane ("reconv")[6], and *iii)* the case in which there are no failures ("no-fail").

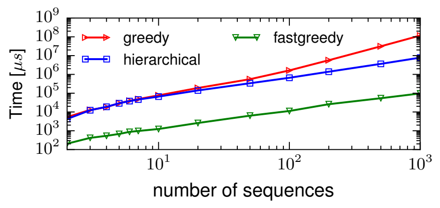
**Simulation reproducibility.** We used ns3 [38] to evaluate the impact of different FRR primitives. To make our simulations realistic, we leverage the publicly-available codebase of the state-of-the-art datacenter load balancer, *i.e.*, Hermes [22]. We inherit the same datacenter topology, workloads, traffic generators, routing schemes, and transport protocols. We implement different FRR primitives and FRR mechanisms on top of this code and evaluate their performance. Our code is released to the public and fully reproducible [16].

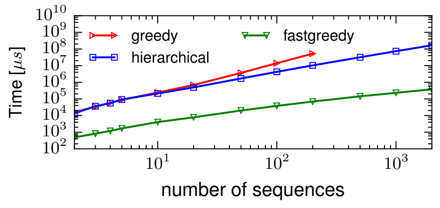
[6]In reality, reconvergence may take up to hundreds of milliseconds or even seconds to happen [37]. During this time, packets arriving at the failed link would be dropped.


(a) Sequence size = 8.


(b) Sequence size = 16.


(c) Sequence size = 32.


(d) Sequence size = 8.


(e) Sequence size = 16.

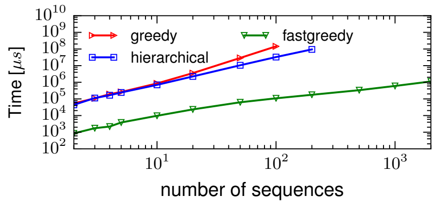
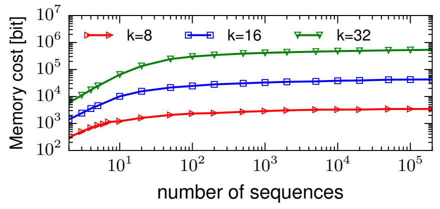
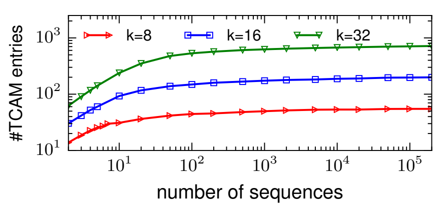

(f) Sequence size = 32.

Figure 11: Comparison of TCAM memory bits and processing times with respect to the number of sequences.


(a) Memory consumption in TCAM bits.


(b) Memory consumption in TCAM entries.

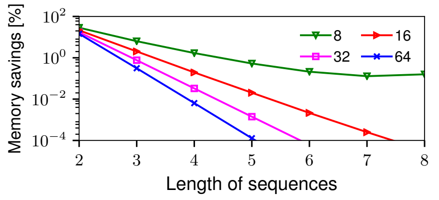

(c) Memory savings for different number of ports.

Figure 12: (a-b) Fast-Greedy with FRR sequences of size $k$. (c) Memory savings.

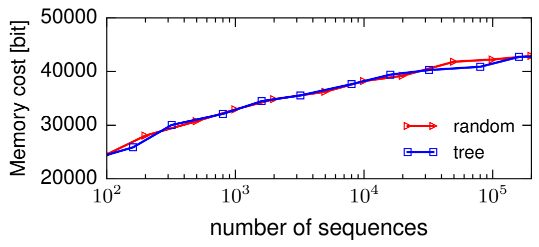


Figure 13: (a-b) Fast-Greedy with FRR sequences of size $k$. (c) Comparing random and tree [23] sequences.

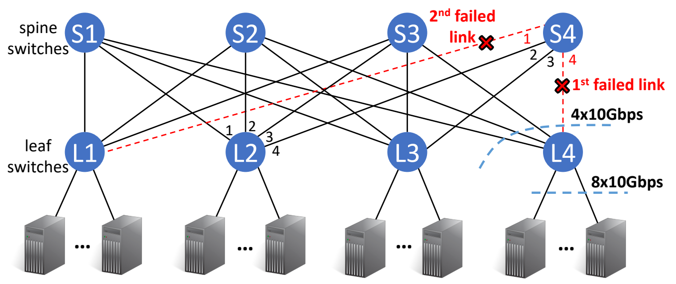


Figure 14: Topology used for simulated evaluation.

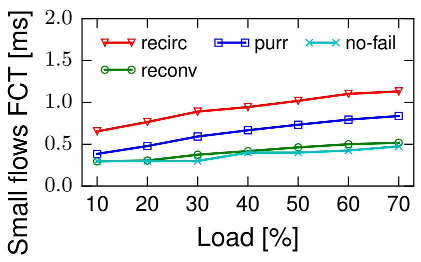
**Topology.** We instantiate 4 leaf and 4 spine switches (see Fig. 14). Each leaf switch interconnects 8 servers. All links are 10 Gbps. The switching fabric has a 2 : 1 oversubscription factor [39], [22]. The buffer size is 100 packets per port. The maximum packet size is 1.3 KB. The leaf-spine and leaf-server link delays are 10 $\mu s$ and 1 $\mu s$, respectively.

**Routing and congestion control.** We rely on the widely adopted Valiant Load Balancing (VLB) routing mechanisms to forward traffic in the datacenter [40]. Each flow between two servers connected to two distinct leaf nodes is forwarded to a random spine node and then directly to the destination leaf node. VLB has been widely implemented using OSPF/ ECMP [41], which splits flows using a deterministic hash-based equal traffic splitting mechanism.

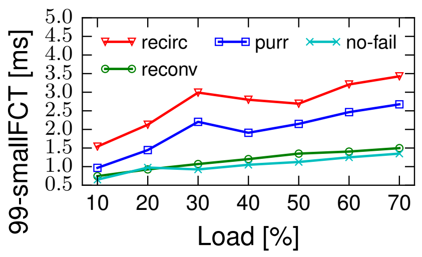
**Transport protocols.** We use DCTCP [42] as the congestion control mechanism. DCTCP supports low-latency and high-throughput communication. We use the same parameters used in Hermes, setting the ECN threshold to [15, 15] packets.

**FRR mechanism: F10 [9].** We implement F10 as the FRR mechanism. F10 is the state-of-the-art FRR mechanism in datacenter networks. In a datacenter with $k$ links between a leaf node and the above spine layer, F10 is capable of tolerating up to $k-1$ link failures, *i.e.*, packets are guaranteed to reach their correct destination without entering transient forwarding loops or being dropped. F10 relies on circular FRR sequences, which we implement on all the network nodes. For example, in Fig. 14, the circular sequence at node $S4$ is $\langle 1, 2, 3, 4\rangle$, which means that when both links $(L4, S4)$ and $(L1, S4)$ fail, a packet that should be sent on port 4 would instead be sent on port 2, which is the first non-failed port in the circular sequence. When the packet is received at node $L2$, we apply again circular FRR forwarding and the packet is sent to $S1$, which, in turn, forwards it to the correct destination.

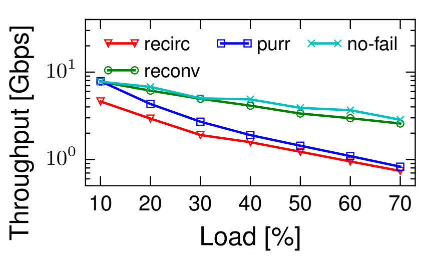
**Workloads.** We use two empirically-derived realistic workloads: *i.e.*, *web-search* [42] and *data-mining* [40]. Both distributions are heavy-tailed, with the data-mining workload being more skewed, thus causing higher imbalances due to ECMP. The traffic generator is based on the work in [43], which generates flows between inter-cluster hosts according to a Poisson distribution and the given network load, which



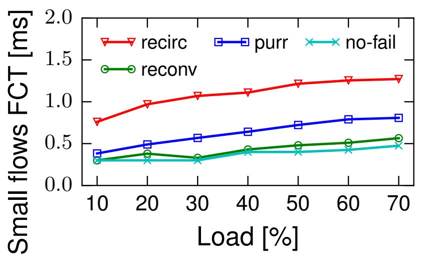
(a) Data-mining, 1 link failure.



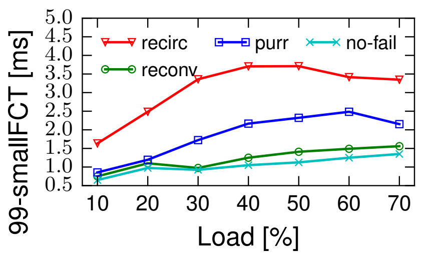
(b) Data-mining, 1 link failure.



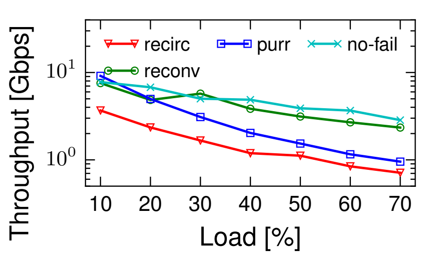
(c) Data-mining, 1 link failure.



(d) Data-mining, 2 links failures.



(e) Data-mining, 2 links failures.



(f) Data-mining, 2 links failures.

Figure 15: Comparison between PURR and RECIRCULATION FRR primitives under 1 and 2 link failures.

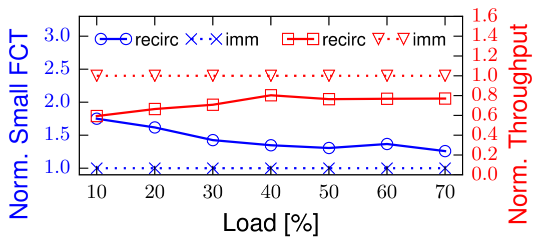
ranges between 10% and 70%, a typical network utilization in a datacenter [43]. We distinguish between small flows (*i.e.*, size $\leq$ 100 KB) and large flows (*i.e.*, size $\geq$ 10 MB).

**Metrics.** For each network load, workload, and FRR primitive, we simulate 4 seconds of traffic. For the RECIRCULATION and PURR FRR primitives, we fail one or two links after 500 ms from the start of the simulation. This effectively simulates a failure of 3.5s, which is a worst-case scenario in datacenter networks [**?**]. For the OSPF reconvergence approach, we fail one or two links at time zero and immediately recompute the optimal routing. We measure the FCT, defined as the time difference between the last received packet and the first "time-scheduled" sent packet, for all the flows that end after 500 ms. We use the OSPF reconvergence simulation to compute an upper bound on the optimal FCT achievable by an FRR primitive. For each setting, we ran a minimum of 40 simulations and compute the average and 99'th percentile of the FCT and flow throughput.

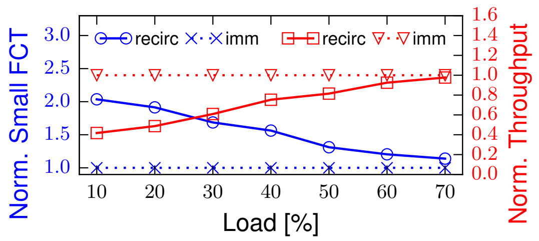
**Modeling packet recirculation in ns3.** When we recirculate a packet in a PDP, the packet moves back to the ingress pipeline, thus congesting the ingress buffer. Since ns3 does not model ingress buffers, we add one "virtual ingress buffer" node in front of each port. We set all latencies to zero so as to mimic an ingress buffer attached to the pipeline. We collaborated with a network engineer from a manufacturer of hardware PDPs to make the model general without breaching our non-disclosure agreement.

**PURR dramatically improves the FCT of small flows.** We ran our simulations for the data-mining workload using the aforementioned setting and we collected our results in Fig. 15. With low network loads, *e.g.,* 10%, and one link failure (see Fig. 15a) we observe that our FRR primitive reduces the FCT of the small flows from the 653 $\mu s$ with packet recirculation to 384 $\mu s$. This means that the FCT *overhead* introduced by FRR compared to the 295 $\mu s$ of the reconverged approach is reduced by a factor of 4.3x. The main reason packet recirculation incurs a higher FCT at low network loads is the packet recirculation operation, which requires to traverse the forwarding pipeline (including its possibly congested ingress buffer) a second time. Even at higher loads, the PURR FRR primitive reduces the FCT overhead by a factor of 2x compared to recirculating a packet. At higher network loads, we note that PURR performs worse than the control plane approach. This happens because PURR routes packets to a core node that does not have a valid downward path towards the destination. This means the traffic has to be rerouted to a leaf node and bounced back to another core node with a valid downward path. Consequently, PURR creates more congestion on the buffers at the core node adjacent to the failed link, which increases the FCT of the small flows. The control plane approach instead routes these affected flows of traffic directly to a core node with a non-failed downward path to the destination. With two link failures (Fig. 15d) the trends are similar though the improvements at 10% and 70% network loads reach 5.5x and 2.8x as the buffers become even more congested than with one single failure.

**PURR guarantees near-optimal throughput at low network loads.** We measure the throughput of the largest flows in the network and compare it among the same four approaches in Fig. 15c and Fig. 15f under 1 and 2 failures, respectively. The throughput of the large flows is computed as the ratio between the amount of all the received bytes and the sum of the FCTs. We note that at 10% network load, PURR achieves the same throughput of the reconverged approaches, approaching 8 Gbps, a factor of 2x higher than with packet recirculation. As the network load increases, the throughput of PURR quickly decreases, faster than in the reconverged setting. This sharper drop of throughput can be explained by the simple fact that at higher load, the impact of going through a node with a lower available bandwidth is exacerbated. We observe one peculiar result that seems counter-intuitive. We note that we cannot compare the performance between one and two link failures, as both the set of affected flows as well as the number of flows



(a) Web-search, 1 link failure.



(b) Web-search, 2 link failures.

Figure 16: FCT and throughput of the large flows normalized with respect to the PURR FRR primitive.

reaching the node with two failed links are different.

For instance, with two failures, the amount of traffic received by leaf node $L4$ is half than with a single failure.

**PURR improves performance on different workloads.** We run simulations using the web-search [40] workload and measure the FCT of the small flows and the throughput of the large flows. Fig. 16 quantifies the performance drop of RECIRCULATION normalized with respect to PURR. As for the data mining workload, we observe that the benefits of PURR are higher at low network loads while they decrease as the network becomes more congested and there is less spare bandwidth for rerouting the affected flows.

## *C. FPGA Evaluation*

Here, we answer the following question: "*How many resources do we need to implement* PURR *on an FPGA chip?*" Table I compares the resource utilization between a layer2 switch and the same system augmented with our primitive on NetFPGA-SUME. *FRR16*, *FRR32* and *FRR64* represent the case when PURR needs 16, 32, and 64 entries in the TCAM, respectively. Such entries can be used to enable different FRR sequences for the selected output port or to allow a single FRR sequence in a system with a larger number of ports. Considering the FRR16 case, PURR impacts only 0.07% of the total available resources of the Slice Lookup Tables (LUTs). The impact grows almost quadratically in the number of TCAM rules. Other resources, *i.e.*, Flip Flops and BRAM, are not affected. This is because Slice LUTs are the main type of resources being used to instantiate TCAMs on FPGAs.

| Project | Slice LUTs | Flip Flops | BRAM |
|---|---|---|---|
| Switch | 43212 | 64811 | 204 |
| Switch + FRR16 | 43523 | 64845 | 204 |
| Switch + FRR32 | 44304 | 64901 | 204 |
| Switch + FRR64 | 46476 | 65006 | 204 |

Table I: HW switch augmented with PURR

## VI. Frequently Asked Questions

**Does PURR support any FRR mechanism?** Yes! To the best of our knowledge, PURR supports any *deterministic* FRR mechanism in which the modifications of the header and the selected output port only depend on the packet header itself, the state currently stored on the switch (*e.g.*, its registers, tables), and the FRR sequence to be applied to the incoming packet. If the selected outgoing port depends on the specific set of failed ports, PURR cannot encode such FRR functions. We are however not aware of any existing FRR scheme that would not be implementable in PURR. As an example, consider MPLS FRR [?], [?] where the header rewriting operation, *i.e.*, addition of a label on the stack, only depends on the selected egress port and the current label. In this case, when a packet arrives and its outgoing port is down, PURR selects the first active outgoing port and the egress pipeline will add the correct label identifying the backup path on that interface for that specific packet. We note that restoration mechanisms requiring control plane invocation require more complex primitives than PURR, which operates at the data plane level. We leave probabilistic FRR mechanisms (*e.g.,* [27]) as future work.

**Could PURR support selective traffic rerouting when multiple links fail?** Yes! When *many links fail* at one switch, we could use priority queues to reroute the most critical traffic (a small fraction of the overall traffic [46]) and drop the rest, based on the remaining capacity. Studying how to reroute the traffic and in which proportions is left as future work.

**How does PURR deal with dynamic updates?** When FRR sequences need to be added or modified at runtime, we need to dynamically update the match-action tables. Three cases can happen (consider Fig. 4b): *i)* the mapping between bits in the `port_set` vector and switch ports remains the same *ii)* the mapping between bits in the `port_set` vector and switch ports changes but its length remains the same *iii)* the mapping between bits in the `port_set` vector and switch ports changes and its length has increased. In case *i)*, we do not have to modify the encoding mapping in $T_2$ and simply modify or add the `port_set` entries in $T_1$. In case *ii)*, we need to update or add the entries in both tables. In the first two cases, the updates can be issued to the P4 runtime, as long as the limit on the number of entries is not reached. In the more remote case *iii)*, the width of Table $T2$ has to be increased and the answer clearly depends on the support from the target device. For instance, techniques on how to partially reconfigure an FPGA in an online manner exist [47]. Similar techniques have been explored to dynamically reconfigure the structure of the P4-based PISA forwarding tables [48], [49]. We note that an operator does not have to recompile the tables if the sequences have non-uniform lengths as long as the mapping allows to implement such sequences. Moreover, if the target architecture imposes certain limits on the TCAM table width, the multi-table approach (discussed in §III-C) can be used for splitting the encoding across multiple tables with a smaller width and length. Finally, we note that one can carefully implement our encoding in a way that any update to the (backup) FRR sequences does not impact the (primary) forwarding rules, thus avoiding any disruption.

**Could PURR be used to implement fast load-balancing forwarding decisions?** Yes! PURR can be generalized to support fast forwarding decisions based on a wide range of conditions. For instance, an operator may be interested in sending a packet to the first active port that has $\leq 50\%$ utilization. We could implement such decision using a vector similar to port `status`, which would however encode the utilization of the ports. We leave this extension as future work.

## VII. Related Work

Connectivity disruptions in networks due to link failures are common and happen in all kinds of networks, from wide-area networks [50], [51] to data center networks [10]. Accordingly, many mechanisms have been developed to provide fast rerouting under failures entirely in the data plane, *e.g.*, [21], [53], [54], [27], [24], [55], [56], [9], [57], [**?**]. FRR mechanisms are also included in MPLS networks [58], [2], IP networks [1] and Openflow [18]. Detecting port failures falls beyond the scope of this paper as it depends on specific hardware support. FRR mechanisms can be generally categorized along different dimensions: *e.g.*, whether they tolerate only a single link/node failure [59], [60], [61] or multiple ones [62], [63]; whether routing tables are static (*e.g.*, [9], [27], [55], [64], [63]) or dynamic (*e.g.*, [65], [57]); whether packet header rewriting (*e.g.*, [57], [56], [66], [62]) or packet duplication (*e.g.*, [67]) is required; whether provide low stretch [44], [27] or maintain relatively low load [68], [69], [70].

This paper *complements* all the above works as our goal is *not* to devise a new robust routing mechanism, but rather a *primitive* which can be used to efficiently implement *existing* mechanisms. Several FRR primitives for quickly rerouting traffic has been proposed, though in different contexts. BGP-PIC [20] and Swift [53] support FRR sequences of size 2. Plinko [63] devised both an FRR mechanism and an FRR primitive to tolerate multiple failures. Unfortunately, the FRR primitive is coupled with the proposed FRR mechanism, thus it cannot support arbitrary FRR sequences. PURR is instead general and supports arbitrary FRR sequences/mechanisms of arbitrary size. Indeed, PURR leaves the choice of which specific failover mechanisms to use to the network operator, but then supports it with a low-latency and compact realization, even tolerating multiple link failures. For example, PURR could be used to realize compact implementations of F10 [9] or [24] which are based on circular FRR sequences. To give another example, PURR supports DDC [57], which provides ideal forwarding connectivity by performing series of link reversal operations dynamically, eventually complementing it with load-aware FRR support as discussed in §VI.

## VIII. Conclusion

This paper presented an FRR primitive for PDPs, which allows to implement existing failover mechanisms with low failover latency and high throughput. Our approach relies on an interesting connection to a classic string manipulation problem for which we also provide new insights, and shows promising results on the PISA-based architectures for which we implemented a prototype. We see our work as a first step towards building highly robust and self-driving programmable networks and believe that it opens several interesting avenues for future research such as finding better heuristics possibly with approximation gurantees. In particular, generalizing our primitive for load-balancing purposes and supporting probabilistic FRR mechanisms seem two attractive future directions.

## Acknowledgements

We would like to thank Andy Fingerhut and Szymon Dudycz for fruitful discussions. This research is supported by the UK's EPSRC under the EARL project (EP/P025374/1), by the European COST Action CA15127, and by the WWTF project WHATIF (ICT19-045).

**Marco Chiesa** is an Assistant Professor at the KTH Royal Institute of Technology, Sweden. He received his Ph.D. degree in computer engineering from Roma Tre University in 2014. His research interests include Internet architectures and protocols, including aspects of network design, optimization, security, and privacy. He received the IEEE William R. Bennett Prize in 2020, the IEEE ICNP Best Paper Award in 2013, and the IETF Applied Network Research Prize in 2012. He has been a distinguished TPC member at IEEE Infocom in 2019 and 2020.

**Roshan Sedar** received the M.Sc. degree in distributed computing from KTH Royal Institute of Technology, Sweden, in 2014. He is currently a researcher at the Telecommunications Technological Center of Catalonia, Spain. He is pursuing his Ph.D. degree at the Polytechnic University of Catalonia, Spain. His research interests include cybersecurity in vehicular communication and next-generation cellular systems, mobile cloud computing, and networked and distributed systems.

**Gianni Antichi** is a Assistant Professor at Queen Mary University of London and Alan Turing Institute fellow. He received his MSc (2007) and PhD (2011) from University of Pisa, Italy. Subsequently, Gianni Antichi worked as postdoc at University of Pisa and University of Cambridge. From 2016 to 2018, he was senior researcher at University of Cambridge. His research interests are at the intersections of networks and systems with a special focus on data plane offloading and end-host networking stacks.

**Michael Borokovich** received B.Sc. (2005), M.Sc. (2009) and Ph.D. (2013) degrees in Communication Systems Engineering from Ben-Gurion University in Israel. The main research topics included fast failover in OpenFlow SDN networks, distributed algorithms, and optimization. Between 2014 and 2015, he was a Postdoc at UT Austin in Texas where he worked on efficient algorithms for distributed graph engines. Between 2015 and 2017, Michael was with AT&T Labs-Research, where he worked on ONAP (Open Network Automation Platform), and VNFs (virtual network functions). Currently, Michael is with Amazon, where he builds innovative SDN solutions for AWS networking.

**Andrzej Kamisiński** is an Assistant Professor at the AGH University of Science and Technology in Kraków, Poland. He received his B.Sc., M.Sc., and Ph.D. degrees from the same University in 2012, 2013, and 2017, respectively. In 2015, Andrzej Kamisiński joined the QUAM Lab at NTNU (Trondheim, Norway) where he worked with Prof. Bjarne E. Helvik and with Telenor Research on dependability of Software-Defined Networks. In summer 2018, he was a Visiting Research Fellow in the Communication Technologies group led by Prof. Stefan Schmid at the Faculty of Computer Science, University of Vienna, Austria. Between 2018 and 2020, he was a member of the Management Committee of the *Resilient Communication Services Protecting End-User Applications From Disaster-Based Failures* European COST Action, and in 2020, a Research Associate in the Networked Systems Research Laboratory at the School of Computing Science, University of Glasgow, Scotland. His primary research interests span dependability and security of computer and communication networks.

**Georgios Nikolaidis** was born in Larisa, Greece in 1983. He received his Diploma in electrical and computer engineering from the National Technical University of Athens in 2006, his M.Sc. in Data Communication Networks and Distributed Systems from University College London (UCL) in 2008 and his PhD in Computer Science from UCL in 2016. The same year he joined Barefoot Networks (acquired by Intel in 2019), where he works in the Advanced Applications group. His current interests include data plane programmability, in-network computation, telemetry, and congestion control.

**Stefan Schmid** is a Professor at the University of Vienna, Austria. He received his MSc (2004) and PhD (2008) from ETH Zurich, Switzerland. Subsequently, Stefan Schmid worked as postdoc at TU Munich and the University of Paderborn (2009). From 2009 to 2015, he was a senior research scientist at the Telekom Innovations Laboratories (T-Labs) in Berlin, Germany, and from 2015 to 2018 an Associate Professor at Aalborg University, Denmark. His research interests revolve around algorithmic problems of networked and distributed systems, currently with a focus on self-adjusting networks (related to his ERC project AdjustNet).