PURR: A Primitive for Reconfigurable Fast Reroute

(hope for the best and program for the worst)

Marco Chiesa
KTH Royal Institute of Technology

Roshan Sedar
Universitat Politècnica de Catalunya

Gianni Antichi
Queen Mary University of London

Michael Borokhovich
Independent Researcher

Andrzej Kamisiński
AGH University of Science and Technology in Kraków

Georgios Nikolaidis
Barefoot Networks

Stefan Schmid
Faculty of Computer Science
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 arbitrary network-wide FRR mechanisms. Our simulations and prototype implementation (on an FPGA and Tofino) show that PURR improves TCAM memory occupancy by a factor of 1.5x—10.8x 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 2.8x—5.5x and 1.2x—2x, respectively, compared to approaches based on recirculating packets.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Network reliability**; **Programmable networks**.

KEYWORDS

programmable networks, network robustness, fast reroute, fast failover, shortest common supersequence

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365410

ACM Reference Format:
Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute: (hope for the best and program for the worst). In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359989.3365410

1 INTRODUCTION

Emerging applications, e.g., in the context of business [21] and entertainment [57], 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 [5, 39, 64] 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 significantly: as soon as a switch detects a failure, i.e., defective link or port, it can quickly detour the affected packets using its own local backup rules.

Networking equipment manufacturers have so far integrated the selected 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 [14], 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 [41, 60], traffic load-balancing [40], and many others [8]. However, little is known today about how to implement arbitrary 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.*, [45]). This is challenging and involves multiple goals:

- ***Flexibility:*** We aim to devise an *FRR primitive* that supports arbitrary FRR mechanisms robust to *single* and *multiple* link failures [26, 49]. FRR mechanisms deal with the computation of primary and backup forwarding rules. The scope of this work is to support the fast transition from primary to backup rules at the individual switch level.
- ***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 [4].

In this paper we propose a new ***FRR primitive***, PURR[1], that serves as a building block for implementing any arbitrary FRR mechanisms while meeting the above requirements. At the heart of PURR lies a technique that *avoids recirculating packets* through the entire packet forwarding pipeline in search of an active (non-failed) port, which would lead to worsened performance, *i.e.*, higher latency and lower throughput. In order to provide memory efficiency, PURR leverages an intriguing 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 arbitrary 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 SCS. 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 adopted as a building block for implementing arbitrary 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 microbenchmarks 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 [20], a Tofino switch [9], and an FPGA [74].

Our code is available to the public and fully reproducible [28].

[1]PURR stands for "a **P**rimitive for reconfig**U**rable fast **R**e**R**oute".

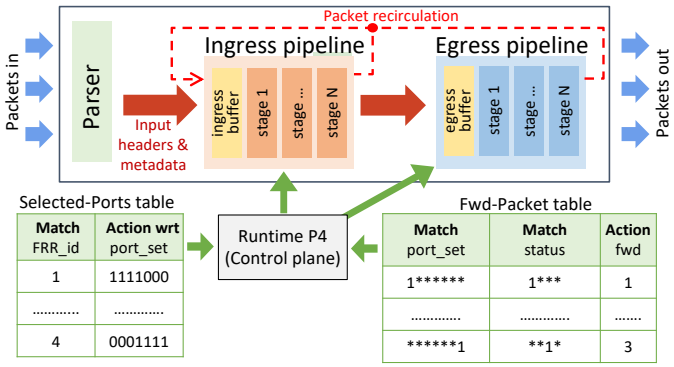


Figure 1: PISA abstraction with PURR pipeline.

2 BACKGROUND AND MOTIVATION

P4 background. P4 [14] 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 [19], *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 [19]. 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 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, SRAM, and DRAM memories contained within each stage of the pipeline. Clearly, different memory types strike different trade-offs in terms of cost, energy consumption, and latency. TCAM memories *support* 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.

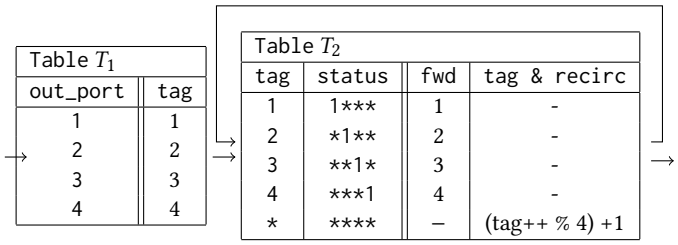
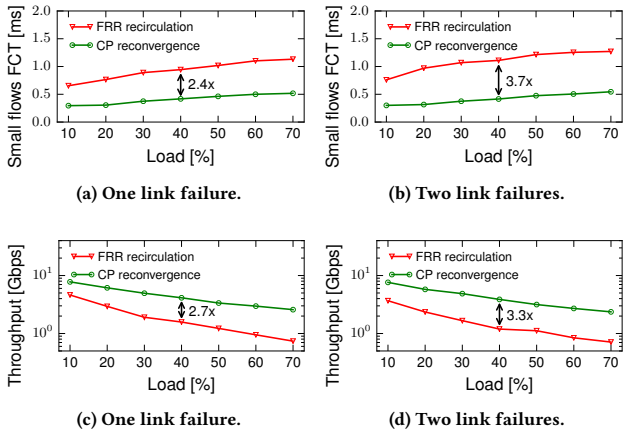


Figure 2: A packet recirculation forwarding table.



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

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

Figure 3: Packet recirculation performance analysis.

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 [24]. 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 extensive discussions with P4 developers, the implementation of FRR sequences in P4 is today left to the operator [53]. We note that FRR primitives devised in different contexts (*e.g.*, BGP-PIC [10, 18]) 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 failure detection. 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 of 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 should try to forward the packet, *i.e.*, the `tag` field, and increase this value if the currently 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^{th} bit in `status` represent the i^{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^{th} entry in the second table T_2 and be forwarded on port 4. When port 4 fails, the same packet will now match the 5^{th} entry in T_2. This will modify `tag` to 1 and the packet will be recirculated, now matching the 1^{st} entry and being routed on port 1.

Packet recirculation degrades flow completion time. There are (potentially) few drawbacks with the above implementation: when a packet is recirculated, *i)* it can add an additional bandwidth overhead on the switch capacity, resulting in a sort of "self-induced incast" on the ingress buffer, *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 better understand the impact of recirculating packets in a concrete setting, 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 existing ns3 implementations from the state-of-the-art datacenter load-balancing codebase (*i.e.*, Hermes [71]) and implemented the F10 [45] state-of-the-art FRR mechanism on top of it. The topology is a leaf-spine datacenter topology with two tiers, the congestion control is DCTCP, and the routing is OSPF/ECMP. 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").[2] 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.

[2]Refer to Sect. 5 for detailed information about the datacenter setting.

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.

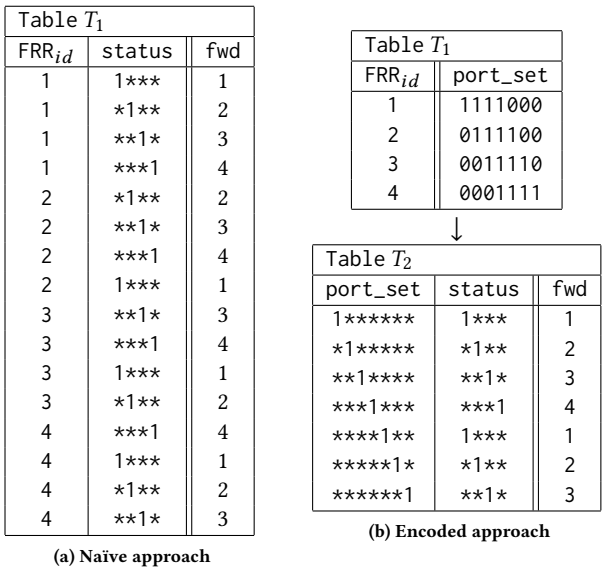
A TCAM-based parallel search to the rescue! A P4 compiler can encode a set of if-else statements within a TCAM memory, which allows to perform the active-port search in parallel. We present one naïve encoding approach in Fig. 4a where we realize the same circular FRR sequences of the packet recirculation case with one single TCAM lookup. One can 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. This approach is similar to the packet recirculation one but we now find the first active port in "one shot", *i.e.,* 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 FRR circular sequences, we need 5760 TCAM entries and $\sim$ 130 kbit of TCAM space, which is already two order of magnitude larger than what is available in today's high-performance PDPs [4]. 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?"

3 A PRIMITIVE FOR FAST REROUTE

We now consider the problem of *encoding* an arbitrary set of FRR sequences into a match-action TCAM-based packet processing pipeline. We first discuss how to realize a specific set of FRR sequences (which we call "circular" FRR sequences), that capture a wide variety of FRR mechanisms that have been proposed to cope with multiple network failures [12, 16, 45]. Finally, we devise a heuristic that efficiently encodes any type of arbitrary FRR sequences into TCAM memories.



(a) Naïve approach

(b) Encoded approach

Figure 4: TCAM encodings of a circular FRR sequence.

3.1 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 and reconverge the network data 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. The 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 typically leverage fast memories for ternary matching (*i.e.*, TCAMs). Even among physical switches with TCAM support there are differences to be taken into account. As an example, the Intel FlexPipe [55] architecture does not support arbitrary width sizes for TCAM tables, a functionality that is supported in the RMT (Reconfigurable Match Tables) architecture [15]. 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.* Each pipeline architecture consists of a certain number of stages through which packets are

being classified and modified. Certain 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 Sect. 2, 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 [4].[3] In the FlexPipe architecture, there are only two stages with 12K entries each. In each stage s, the amount of TCAM memory (in bits[4]) 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 will allow efficient realization of any set of FRR sequences. We already explained in Sect. 2 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 [12] or circular-arborescence [17]) our proposed primitive will allow deploying them in a way that reduces the amount of TCAM memory required.

3.2 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 [45], arc-disjoint arborescences [17], and graph-traversals [12]. 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 set of FRR sequences $\{F_1, F_2, F_3, F_4\}$ are circular.

Encoding circular FRR sequences. We already described a naïve approach for encoding circular FRR sequences in Sect. 2, 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 in the switch (and hence, the length of an FRR sequence). Let us now propose a more efficient way of 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.

The 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 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 down and ports 2 and 3 are up, 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 memories compared to a naïve approach. Let us now analyze the TCAM space required to encode a set of n circular FRR sequences, each sequence having length k (notice that there are at most k such sequences, i.e., $n \leq k$). 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 result gives an order of magnitude improvement over the naïve approach which requires $nk(k + \log n) = O(k^3)$ TCAM bits. Notice also that table T_1 does not require ternary matches and therefore can be implemented in SRAM, saving the limited and expensive TCAM space even more.

3.3 A Primitive to Implement Them All

We now introduce and tackle the general problem of *encoding* 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. We tackle this problem by generalizing 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, the encoding problem boils down to finding the shortest sequence that contains all the given sequences as subsequences (*i.e.*, skipping elements is allowed).

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) [30] problem, *where no repetitions are allowed.* In the SCS problem, the input is a set of sequences

[3] Also based on private communication with vendors.
[4] For simplicity, we use the "bit" terminology as opposed to the more correct "trits" one, which captures the ternary nature of the TCAM elements.

$\mathcal{S} = \{S_1, \ldots, S_k\}$ 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 [56], and also hard to approximate within polylogarithmic factors [37].

Unfortunately, this is not the case: we state this insight as a theorem as the result is of independent interest.

Theorem 3.1. *The SCS problem without repetitions is NP-hard to optimize and approximate.*[5]

The proof follows from a careful analysis of the proof in [37] for the general SCS problem (not repeated here due to space constraints). In no step during this proof, are any repetitions needed.

The dynamic programming building block: DPSCS. We first discuss a well-known technique used to solve the SCS problem optimally based on Dynamic-Programming [69], 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 an ideal baseline to compare our proposed heuristic and to deal with arbitrary number of sequences. The input to our problem is a set $\mathcal{F} = \{F_1, \ldots, F_n\}$ 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 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 1), which strikes a different tradeoff in terms of fast running time and reasonably good accuracy. At each iteration, we trim the left-most element 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 FRR sequence. In Fig. 5, we show an example of Fast-Greedy with four sequences F_1=⟨2 3 1 0⟩, F_2 =⟨0 2 1 3⟩, F_3=⟨3 0 2 1⟩, and F_4=⟨1 0 2 3⟩. 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 < 2 0 3 1 0 2 1 3 >. 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. We now consider the problem in which the FRR encoding can be realized across multiple tables instantiated

[5] 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($2^{\text{polylog } n}$).

Algorithm 1 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* :=⟨⟩
(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* ∪ < a >

(3) **return** *currscs*

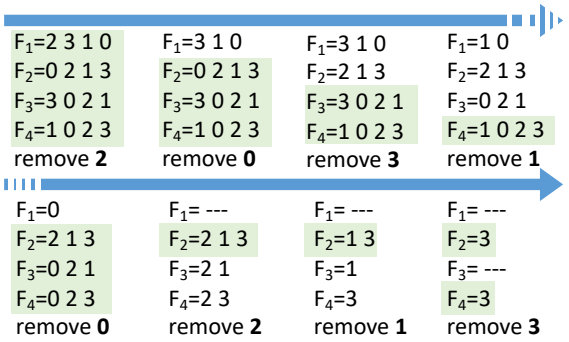


Figure 5: Fast-Greedy example.

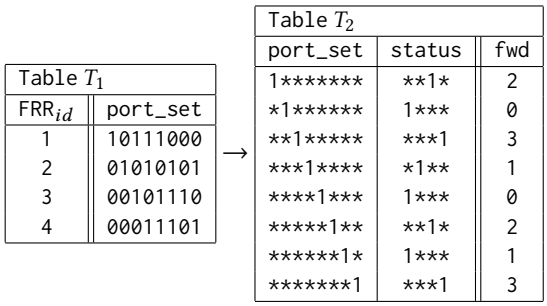


Figure 6: Fast-Greedy TCAM implementation.

Algorithm 2 Definition of MultiTable-SCS (MT-SCS).

Function input: A set $\mathcal{F} = \{F_1, \ldots, F_k\}$ of FRR sequences
(1) Let $S = \{\}$, add $\{F_1\}, \ldots, \{F_k\}$ into S, and let f = True
(2) Repeat until f is True
  (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 $cost(S) \leq cost(S')$, f =False; else $S = S'$
(3) **return** S

in the same stage of the pipeline, which is possible on today's programmable switches [38]. This potentially allows us to build even more compact representations of a set of FRR sequences. In some cases, using multiple tables may also be necessary because real 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 (stringology), which is tailored for the specific case of FRR sequences (*i.e.,* no element repetitions).

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 better memory overheads than with a single table.

Finding similar sequences leads us to consider a complementary problem to SCS, *i.e.*, the Longest Common Subsequence (LCS) [48] problem.[6] LCS is renown to be 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.,* $O(nk^2)$).

Theorem 3.2. *The LCS problem without repetitions is polynomial-time solvable.*

This motivates us to consider LCS as a way to efficiently group FRR sequences into different tables. In MultiTable-SCS (Alg. 2), 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, we repeat the procedure. We stop it otherwise and return the set partitioning, each set corresponding to a table encoding.

4 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 [20] 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 [16], as well as the Depth First Search (DFS), Breadth First Search (BFS) and the rotor router mechanisms in [11]. 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 is a simple operation. It simply entails incorporating 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 the PURR prototype on the NetFPGA-SUME platform [74], which is a PCIe adapter card with $4x10$ Gbps Ethernet interfaces and a large 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.

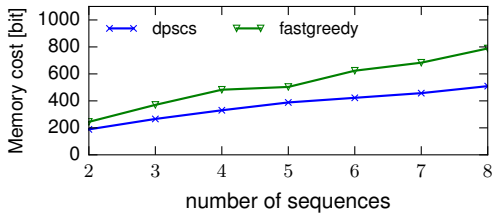
5 EVALUATION

We now assess the performance of the algorithms introduced in Sect. 3 for encoding a set of FRR sequences into a TCAM memory. We evaluate the algorithms along two dimensions: the amount of memory needed to encode the FRR sequences in memory bits and their running time. This first part of the evaluation focuses on a single switch that gets a set of FRR sequences as input and computes an encoding of these sequences in a TCAM memory. In the second part of the evaluation, we set up a datacenter Clos network and implement the state-of-the-art FRR mechanism for Clos networks, *i.e.,* F10 [45], using circular FRR sequences. We then run simulations in *ns3* to study the impact of using PURR w.r.t. an approach based on recirculating packets.

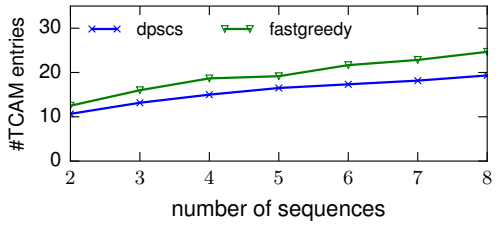
5.1 FRR Encoding

In this section, we answer the following main question: "*How much TCAM memory (in bits and entries) do we need to implement a given set of FRR sequences?*". We implement DPSCS and Fast-Greedy in Python and consider three different dimensions: *i)* we vary the number of FRR sequences n, *ii)* we vary 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.

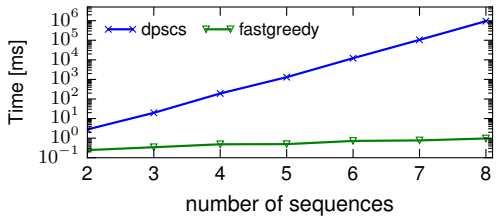
[6] Note that, formally, LCS is not the dual problem of SCS.



(a) Memory consumption in TCAM bits.

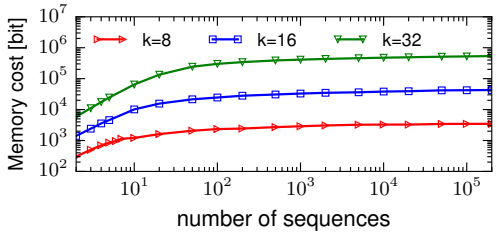


(b) Memory consumption in TCAM entries.

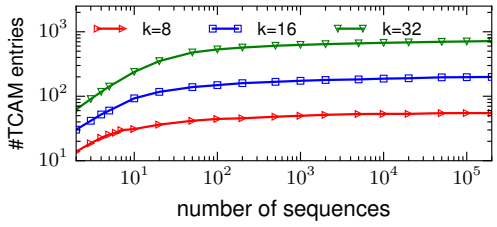


(c) Processing time.

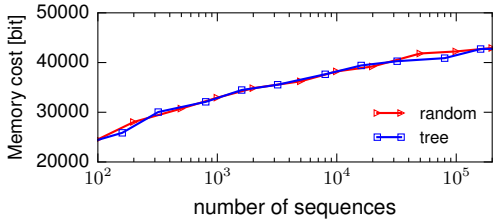
Figure 7: Comparison of Fast-Greedy with respect to the optimum. The size of the sequences is set to 7.



(a) Memory consumption in TCAM bits.



(b) Memory consumption in TCAM entries.



(c) Disjoint-tree vs. random sequences.

Figure 8: (a-b) Fast-Greedy with FRR sequences of size k. (c) Comparing random and tree [16] sequences.

Encoding FRR sequences is crucial in high port density switches. We first evaluate the Naïve approach described in Fig. 4a and compare it with our encoding-based mechanism described in Fig. 4b. The results are based on the calculations described in Sect 3.2. We consider the broad family of FRR mechanisms (*e.g.*, F10 [45], DFS [12], basic arc-disjoint spanning trees [16]), 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 target with 64 ports would require 327 KB of TCAM to implement 10 circular FRR sequences. This corresponds to 2 entire pipeline stages on the RMT architecture and roughly 5 stages in real-world programmable data planes [4]. An encoded approach would merely require 30 KB, one tenth of the TCAM memory contained in a single stage of the RMT architecture [15].

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. 7a and Fig. 7b 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. 7c. 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 compresses hundreds of thousands of FRR sequences within limited memory. We show in Fig. 8a and Fig. 8b the amount of memory in bits and the number of entries required to implement a given set of FRR sequences. Our results show that 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 amount of 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.

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 [16]. We

quantify the memory requirements of an arc-disjoint FRR mechanism, called *tree*, in Fig. 8 deployed on Jellyfish [59] 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 implementing FRR the rerouting decision made in Log-Basic 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 [15].

5.2 Datacenter Simulations

In this section, we answer the following main question: *How does the flow completion time (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 our FRR primitive is not specific to datacenter environments but also 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")[7], and *iii)* the case in which there are no failures ("no-fail").

Simulation reproducibility. We used the packet-level *ns3* simulator [1] to evaluate the impact of different FRR primitives. To make our simulations realistic and reproducible, we leverage the publicly-available codebase of the state-of-the-art datacenter load balancer, *i.e.*, Hermes [71]. 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 will be released to the public and fully reproducible [28].

Topology. The datacenter topology (see Fig. 9) consists of 4 leaf and 4 spine switches. Each leaf switch interconnects 8 servers. All links are 10 Gbps. The switching fabric has a $2:1$ oversubscription factor [2, 71]. 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 μs and 1 μs, respectively.

Routing and congestion control. We rely on the widely adopted Valiant Load Balancing (VLB) routing mechanisms to forward traffic in the datacenter [29]. Each flow of traffic 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 [35], which splits flows of traffic using a deterministic hash-based equal traffic splitting mechanism.

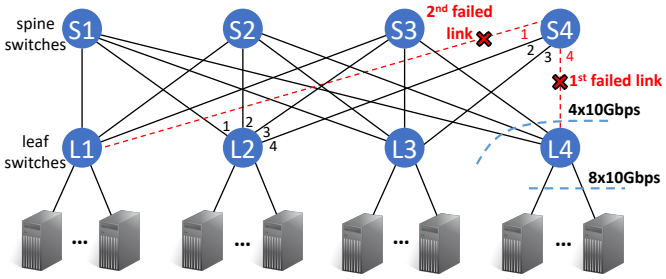


Figure 9: Topology used for simulated evaluation.

Transport protocols. We use DCTCP [3] as the congestion control mechanism. DCTCP supports low-latency and high-throughput communication. We use the same parameter of Hermes, setting the ECN threshold to [15, 15] packets.

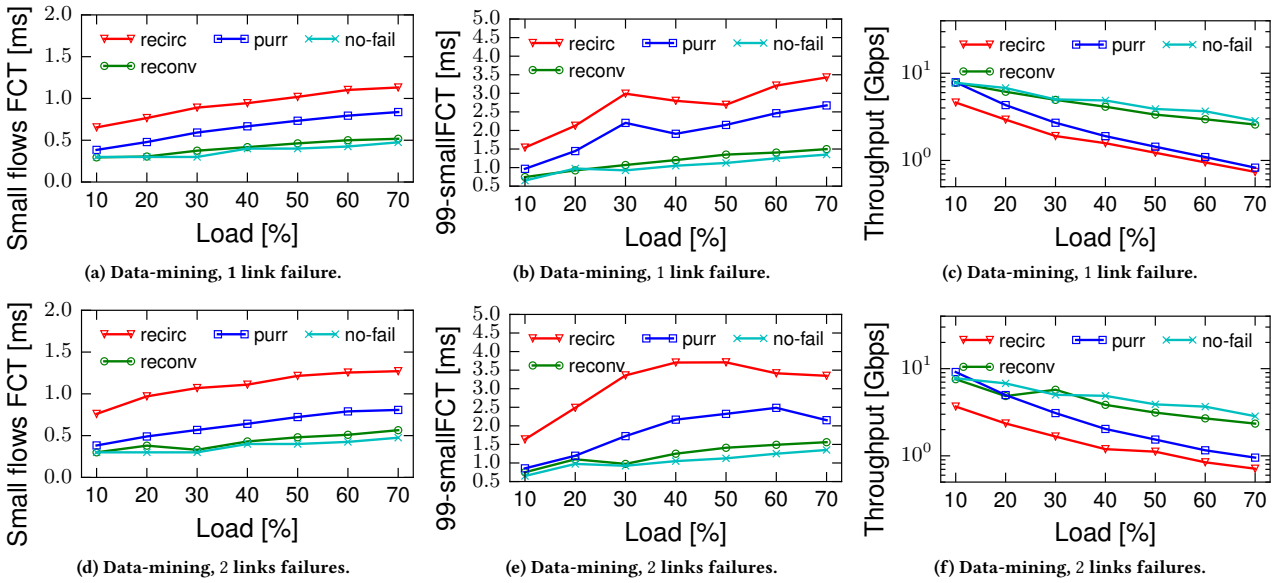
FRR mechanism: F10 [45]. We implement F10 as the FRR mechanism in our topology. F10 is the state-of-the-art FRR mechanism in datacenter networks. In a datacenter with k links between a leaf node and the above spine layer, F10 is capable of tolerating up to $k-1$ link failures, *i.e.,* packets are guaranteed 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. 9, the circular sequence at node $S4$ is $< 1, 2, 3, 4 >$, 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* [3] and *data-mining* [29]. 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 [7], which generates flows of traffic between inter-cluster hosts according to a Poisson distribution and the given network load, which ranges between 10% and 70%, a typical network utilization in a datacenter [7]. 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 For the OSPF reconvergence approach, we fail one or two links at time zero and immediately recompute the optimal OSPF routing. We measure the Flow Completion Times[8] (FCTs) for all the flows that ends after 500ms. 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

[7] In reality, reconvergence may take up to hundreds of milliseconds or even seconds to happen [58]. During this time, packets arriving at the failed link would be dropped.

[8] Defined as the time difference between the last received packet and the first "time-scheduled" sent packet.



(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.

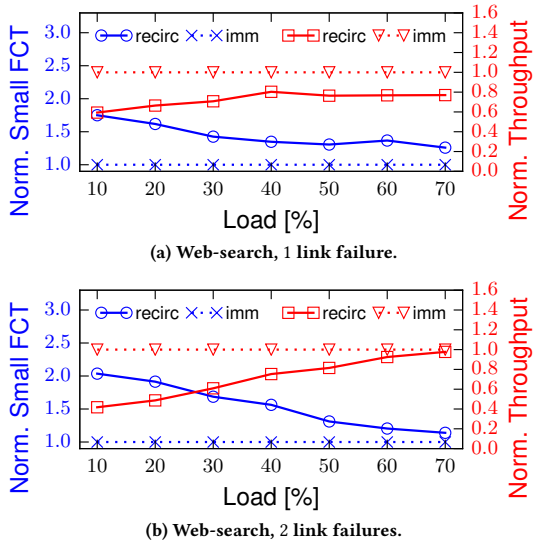
Figure 10: Comparison between PURR and RECIRCULATION FRR primitives under 1 and 2 link failures.

40 simulations and compute the average and 99'th percentile of the FCT and flow throughput.[9]

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 pipeline. We collaborated with one network engineer from a manufacturer of hardware PDPs to validate our model while trying to keep the model as general as needed to guarantee non-disclosure agreement. We defer the reader to App. A for further details.

PURR dramatically improves the flow completion time (FCT) of the small flows. We ran our simulations for the data-mining workload using the aforementioned setting and we collected our results in Fig. 10. With low network loads, *e.g.,* 10%, and one link failure (see Fig. 10a) we observe that our FRR primitive reduces the FCT of the small flows from the 653 μs with packet recirculation to 384 μs. This means that the FCT *overhead* introduced by FRR compared to the 295 μs 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. 10d) 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.

[9]We ran simulations for the equivalent of roughly 10000 hours (more than one year) of computing time on a 2.60 GHz machine.



(a) Web-search, 1 link failure.

(b) Web-search, 2 link failures.

Figure 11: FCT and throughput of the large flows normalized with respect to the PURR FRR primitive.

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. 10c and Fig. 10f 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 flow completion times. 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 seem counter-intuitive. We note that we cannot compare the performance between one and two link failures as the set of affected flows, as well as the number of flows reaching the node with two failed links, is different. For instance, with two failures, the amount of traffic received by leaf node $L4$ is 50% smaller than with one single failure.

PURR improves performance on different workloads. We run simulations using the web-search [29] workload and measure the FCT of the small flows and the throughput of the large flows normalized with respect to PURR. Fig. 11 quantifies the performance drop of RECIRCULATION normalized with respect to PURR. As for the datamining 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.

5.3 FPGA Evaluation

In this section, we answer the following question: "*How many resources do we need to implement PURR on an FPGA chip?*" Table 1 compares the resource utilization between a simple NetFPGA-SUME switch and the same system augmented with our primitive. *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. The 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 1: HW switch augmented with PURR

6 FREQUENTLY ASKED QUESTIONS

Does PURR support any FRR mechanism? Yes! To the best of our knowledge, PURR supports any *deterministic* FRR mechanism proposed in the literature for datacenters and WAN networks, including load-aware ones [23]. PURR receives as input a set of FRR sequences that needs to be implemented into the networking devices similarly to the OpenFlow fast reroute groups [51]. As long as an FRR mechanism describes its primary and backup forwarding behaviour as a set of primary and backup ports, PURR can encode such a mechanism into the dataplane pipeline. 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.,* [17]) as future work.

Could PURR support selective traffic rerouting when multiple links fail? Yes! When *many links fail* at one switch, we could leverage priority queues to reroute the most critical traffic — a small fraction of the overall traffic [36] — and drop the rest, based on the available remaining capacity. Studying how to reroute the traffic and in which proportions is left as future work.

How does PURR deal with dynamic updates? In the cases when FRR sequences need to be added or modified at runtime, we need to dynamically update the match-action tables. We divide dynamic updates into three cases (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 the 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 [66]. Similar techniques have been explored to dynamically reconfigure the structure of the P4-based PISA forwarding tables [72, 73]. 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 Section 3.3) 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! We believe PURR can be generalized to support fast forwarding decisions based on a wide range of programmable 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.

7 RELATED WORK

Connectivity disruptions in networks due to link failures are common [44, 47, 50, 65] and happen in all kinds of networks, from wide-area networks [34, 43] to data center networks [27]. Accordingly, many clever mechanisms have been developed to provide fast re-routing under failures entirely in the dataplane, e.g., [10, 12, 13, 17, 32, 33, 42, 44, 45]. Fast reroute mechanisms are also included in MPLS networks [6, 64], IP networks [5], in Openflow [24], among many others. Detecting port failures falls beyond the scope of this paper as it depends on specific hardware support.

FRR mechanisms can generally be categorized along different dimensions, *e.g.*, whether they tolerate only a single link/node failure [52, 67, 70] or multiple ones [22, 62], or whether routing tables are static (*e.g.*, [13, 17, 45, 61, 62]) or dynamic (*e.g.*, [25, 44]), whether packet header rewriting is required (*e.g.*, [22, 42, 44, 46]) or packet duplication (*e.g.*, [31]), whether provide low stretch [17, 23]or load [54, 63, 68].

This paper *complements* all the above works in that 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 [18] and Swift [33] support FRR sequences of size 2. Plinko [62] 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 instead leaves the choice of which specific failover mechanisms to use to the network operator, but then supports the mechanism by a low-latency and compact realization, even tolerating multiple link failures. For example, PURR could be used to realize compact programmable implementations of F10 [45] or [12] which are based on circular FRR sequences, which paves the way for a seamless implementation on PDPs. To give another example, PURR supports DDC [44], which provides ideal forwarding connectivity by performing series of link reversal operations dynamically, eventually complementing it with load-aware FRR support as discussed in Sect. 6.

8 CONCLUSION

This paper presented an FRR primitive for programmable data planes, which allows to implement existing failover mechanisms without recirculating packets and hence 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 understand our work as a first step building robust and self-driving programmable networks and believe that it opens several interesting avenues for future research. In particular, generalizing our primitive for load-balancing purposes and supporting probabilistic FRR mechanisms seem two attractive future directions.

ACKNOWLEDGEMENTS

We would like to acknowledge our shepherd Cristel Pellser and the anonymous reviewers for their invaluable feedback on this paper. The 1st author would like to thank Andy Fingerhut for discussions on the FRR primitives and the P4 language. The 7th author would like to thank Szymon Dudycz from Wroclaw University for discussions on the SCS problem. This research is supported by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL: sdn EnAbled MeasuRement for alL project (Project Reference EP/P025374/1). The research done by the 5th author is supported by the European Cooperation in Science and Technology (COST) Action CA15127 "Resilient communication services protecting end-user applications from disaster-based failures — RECODIS").

REFERENCES

[1] 2019. ns3 Network Simulator. (June 2019). https://www.nsnam.org/.
[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 503–514. https://doi.org/10.1145/2740070.2626316
[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192
[4] Anonymous. 2019. FlexGate: High-performance Heterogeneous Gateway in Data Centers. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet '19)*. To appear.
[5] Alia Atlas and Alex Zinin. 2008. *Basic Specification for IP Fast Reroute: Loop-Free Alternates.* RFC 5286. RFC Editor. 1–31 pages. https://doi.org/10.17487/RFC5286
[6] François Aubry, Stefano Vissicchio, Olivier Bonaventure, and Yves Deville. 2018. Robustly disjoint paths with segment routing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018.* 204–216. https://doi.org/10.1145/3281411.3281424
[7] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-service Multi-queue Data Centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 537–549. http://dl.acm.org/citation.cfm?id=2930611.2930646
[8] Barefoot. 2018. In-Network DDoS Detection. (November 2018). https://barefootnetworks.com/use-cases/in-nw-DDoS-detection/.
[9] Barefoot. 2019. Tofino: World's fastest P4-programmable Ethernet switch ASICs. (2019). http://barefootnetworks.com/products/brief-tofino/ (accessed on June 26, 2019).
[10] Olivier Bonaventure, Clarence Filsfils, and Pierre Francois. 2007. Achieving Sub-50 Milliseconds Recovery Upon BGP Peering Link Failures. *IEEE/ACM Trans. Netw.* 15, 5 (Oct. 2007), 1123–1135. https://doi.org/10.1109/TNET.2007.906045
[11] Michael Borokhovich, Clement Rault, Liron Schiff, and Stefan Schmid. 2018. The show must go on: Fundamental data plane connectivity services for dependable SDNs. *Computer Communications* 116 (2018), 172 – 183. https://doi.org/10.1016/j.comcom.2017.12.004
[12] Michael Borokhovich, Liron Schiff, and Stefan Schmid. 2014. Provable data plane connectivity with local fast failover: introducing OpenFlow graph algorithms. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 121–126. https://doi.org/10.1145/2620728.2620746
[13] Michael Borokhovich and Stefan Schmid. 2013. How (not) to shoot in your foot with SDN local fast failover. In *Conference on Principles of Distributed Systems (OPODIS)*. Springer-Verlag.
[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890
[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[16] Marco Chiesa, Andrei Gurtov, Aleksander Madry, Slobodan Mitrovic, Ilya Nikolaevskiy, Michael Schapira, and Scott Shenker. 2016. On the resiliency of randomized routing against multiple edge failures. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. Leibniz.

[17] M. Chiesa, I. Nikolaevskiy, S. MitroviÄĞ, A. Panda, A. Gurtov, A. Maidry, M. Schapira, and S. Shenker. 2016. The quest for resilient (static) forwarding tables. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524552

[18] Cisco. 2014. BGP PIC Edge for IP and MPLS-VPN. (2014). https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-3s/irg-xe-3s-book/irg-bgp-mp-pic.html (accessed on June 26, 2019).

[19] P4 Language Consortium. 2017. P4 Language Specification. (May 2017). https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf (accessed on June 26, 2019).

[20] P4 Language Consortium. 2019. Behavioral Model (BMv2). (June 2019). https://github.com/p4lang/behavioral-model.

[21] Yoav Einav. 2019. Amazon Found Every 100ms of Latency Cost them 1% in Sales. In *Gigaspaces*.

[22] Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. 2016. IP Fast Rerouting for Multi-Link Failures. In *Transactions on Networking, Volume: 24, Issue: 5*. IEEE/ACM.

[23] Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. 2019. CASA: Congestion and Stretch Aware Static Fast Rerouting. In *Proc. IEE INFOCOM*.

[24] Open Network Foundation. 2012. Switch Specification 1.3.1. (September 2012). https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[25] Eli M. Gafni and Dimitri P. Bertsekas. 1981. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. In *Transactions on Communications, Volume: 29, Issue: 1*. IEEE.

[26] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 350–361.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *Computer Communication Review, Volume: 41, Issue: 4*. ACM.

[28] GitHub. 2019. PURR Repository. (2019). https://bitbucket.org/marchiesa/purr.

[29] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. *SIGCOMM Comput. Commun. Rev.* 39, 4 (Aug. 2009), 51–62. https://doi.org/10.1145/1594977.1592576

[30] Dan Gusfield. 1997. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press.

[31] Prashanth Hande, Mung Chiang, Robert Calderbank, and Sundeep Rangan. 2009. Network pricing and rate allocation with content provider participation. In *Conference on Computer Communications (INFOCOM)*. IEEE.

[32] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.

[33] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. 2017. SWIFT: Predictive Fast Reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/3098822.3098856

[34] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[35] C. Hopps. 2000. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. RFC Editor. 1–8 pages. https://doi.org/10.17487/RFC2992

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 3–14. https://doi.org/10.1145/2534169.2486019

[37] Tao Jiang and Ming Li. 1995. On the approximation of shortest common supersequences and longest common subsequences. In *Journal on Computing, Volume: 24, Issue: 5*. SIAM.

[38] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*.

[39] Andrzej Kamisiński. 2018. Evolution of IP Fast-Reroute Strategies. In *2018 10th International Workshop on Resilient Networks Design and Modeling (RNDM)*. 1–6. https://doi.org/10.1109/RNDM.2018.8489832

[40] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: congestion-aware load balancing at the virtual edge. In *Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[41] Changhoon Kim, Anirudh Sivaraman, Naga Praveen Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes. In *Industrial demo, ACM SIGCOMM*.

[42] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving convergence-free routing using failure-carrying packets. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[43] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[44] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[45] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A fault-tolerant engineered network. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[46] Suksant Sae Lor, Raul Landa, and Miguel Rio. 2010. Packet re-cycling: eliminating packet losses due to network failures. In *Hot Topics in Networks (Hotnets)*. ACM.

[47] Doug Madory. 2018. Large Outage in Pakistan. (November 2018). http://dyn.com/blog/large-outage-in-pakistan/.

[48] David Maier. 1978. The complexity of some problems on subsequences and supersequences. In *Journal of the ACM, Volume: 25, Issue: 2*. ACM.

[49] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, and Christophe Diot. 2004. Characterization of failures in an IP backbone. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2307–2317.

[50] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. 2008. Characterization of failures in an operational IP backbone network. In *Transactions on Networking, Volume: 16, Issue: 4*. IEEE/ACM.

[51] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. In *Computer Communication Review, Volume: 38, Issue: 2*. ACM.

[52] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. 2007. Fast local rerouting for handling transient link failures. (2007).

[53] P4-dev maling list. 2018. (2018). http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-May/002027.html.

[54] Yvonne Anne Pignolet, Stefan Schmid, and Gilles Tredan. 2017. Load-optimal local fast rerouting for resilient networks. In *Dependable Systems and Networks (DSN)*. IEEE.

[55] Ozdag R. 2012. Intel R Ethernet Switch FM6000 SeriesSoftware Defined Networking. (2012). Intel Corporation.

[56] Kari-Jouko Räihä and Esko Ukkonen. 1981. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science, Volume: 16, Issue: 2*.

[57] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. 2013. Cloud gaming: architecture and performance. *IEEE network* 27, 4 (2013), 16–21.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 183–197. https://doi.org/10.1145/2829988.2787508

[59] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. 2012. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–17. http://dl.acm.org/citation.cfm?id=2228298.2228322

[60] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter detection entirely in the data plane. In *Symposium on SDN Research (SOSR)*. ACM.

[61] Brent Stephens, Alan L. Cox, and Scott Rixner. 2013. Plinko: Building Provably Resilient Forwarding Tables. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII)*. ACM.

[62] Brent Stephens, Alan L. Cox, and Scott Rixner. 2016. Scalable Multi-Failure Fast Failover via Forwarding Table Compression. In *Proceedings of the Symposium on SDN Research*. ACM.

[63] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. 2011. Network architecture for joint failure recovery and traffic engineering. In *Special Interest Group for the Computer Systems Performance Evaluation (SIGMETRICS)*. ACM.

[64] George Swallow, Ping Pan, and Alia Atlas. 2005. Fast reroute extensions to RSVP-TE for LSP tunnels. In *RFC 4090*.

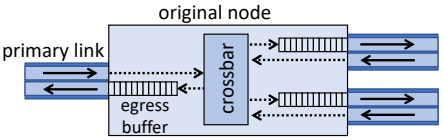
[65] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. 2010. California fault lines: understanding the causes and impact of network failures. *Computer Communication Review, Vol: 40, Issue: 4*.

[66] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Comput. Surv.* 51, 4, Article 72 (July 2018), 39 pages. https://doi.org/10.1145/3193827
[67] Junling Wang and Srihari Nelakuditi. 2007. IP fast reroute with failure inferencing. In *Internet Network Management (INM)*. ACM.
[68] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. 2010. R3: Resilient Routing Reconfiguration. In *Computer Communication Review, Volume: 40, Issue: 4*. ACM.
[69] Wikipedia. [n. d.]. Shortest common supersequence problem. ([n. d.]). https://en.wikipedia.org/wiki/Shortest_common_supersequence_problem (accessed on June 26, 2019).
[70] Baobao Zhang, Jianping Wu, and Jun Bi. 2013. RPFP: IP fast reroute with providing complete protection and without using tunnels. In *International Symposium on Quality of Service (IWQoS)*. IEEE.
[71] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 253–266. https://doi.org/10.1145/3098822.3098841
[72] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 98–111. https://doi.org/10.1145/3281411.3281436
[73] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. ShadowP4: Building and Testing Modular Programs. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM '18)*. ACM, New York, NY, USA, 150–152. https://doi.org/10.1145/3234200.3234231
[74] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (Sep 2014), 32–41. https://doi.org/10.1109/MM.2014.61

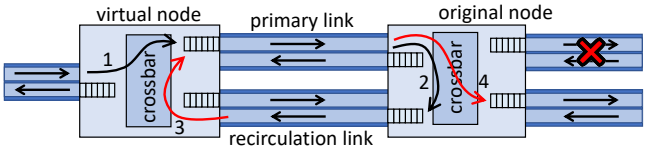
APPENDIX

A PACKET RECIRCULATION IN NS3

Modeling packet recirculation in ns3. The impact of recirculating packets on the forwarded traffic clearly depends on the specific hardware architecture of the considered networking device. The ns3 simulator simplifies the internal architecture of a switch node, which boils down to a crossbar engine (capable of any programmable forwarding function) and a set of egress buffers, one per physical link. See Fig. 12a for a visual representation. Since nodes do not have any ingress buffers, capturing the impact of packet recirculation becomes problematic. In fact, when recirculating a packet, a switch should become congested at the ingress buffer of the port through which a packet is being recirculated (*i.e.*, the one where it was received). This would however never be the case in "vanilla" ns3. Adding a physical loop between two independent ports would not suffice as recirculated packets would be received on a different port and immediately enqueued in the correct egress queue. To reproduce the impact of packet recirculation in ns3, we modeled the ingress buffer of each port on a switch with a "virtual" node that is connected directly to the switch port (see Fig. 12b). We set the latency between the two nodes to zero and the buffer on the virtual node to 100 packets with an ECN threshold of 15 packets. We connect one virtual node to each port of a switch (we show only one virtual node in Fig. 12b), thus virtual nodes simply act as ingress buffers for the original node. The forwarding of a packet is as follows : 1) a packet is received by the virtual node and forwarded to the original node using the original primary link, 2) if the packet hits a failed link, it is recirculated back to the virtual node using the recirculation link, 3) the packet is marked and mirrored back to the original node, and 4) the packet is sent to its backup forwarding egress port on the original node. We assume the latency to process a packet through the pipeline is in the order of



(a) Node representation in ns3.



(b) Packet recirculation implementation (shown only for the primary link) .

Figure 12: Modeling packet recirculation in ns3.

hundreds of nanoseconds (not including the time spent in a buffer). We corroborated the validity of our model with a manufacturer of programmable switches — details are covered by a non-disclosure agreement.