D2R: Policy-Compliant Fast Reroute

Kausik Subramanian
University of Wisconsin-Madison
Madison, WI, USA
sskausik08@cs.wisc.edu

Anubhavnidhi Abhashkumar
University of Wisconsin-Madison*
Madison, WI, USA
abhashkumar@wisc.edu

Loris D'Antoni
University of Wisconsin-Madison
Madison, WI, USA
loris@cs.wisc.edu

Aditya Akella
University of Wisconsin-Madison
Madison, WI, USA
akella@cs.wisc.edu

CCS Concepts

• **Networks** → **Network reliability**; **Programmable networks**.

Keywords

Programmable switches, data plane algorithms, network routing

ACM Reference Format:
Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2021. D2R: Policy-Compliant Fast Reroute. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3482898.3483360

Abstract– In networks today, the data plane handles forwarding—sending a packet to the next device in the path—and the control plane handles routing—deciding the path of the packet in the network. This architecture has limitations. First, when link failures occur, the data plane has to wait for the control plane to install new routes, and packet losses can occur due to delayed routing convergence or central controller latencies. Second, policy-compliance is not guaranteed without sophisticated configuration synthesis or controller intervention. Fast reroute mechanisms in the data plane cannot provide both connectivity and policy-compliance guarantees. We take advantage of the recent advances in fast programmable switches to perform policy-compliant route computations entirely in the data plane, thus providing fast and programmable reactions to failures. D2R provides the illusion of a hierarchical network fabric that is always available and policy-compliant under failures. We implement our data plane in P4 and show its viability in real world topologies.

1 Introduction

With a plethora of performance-sensitive distributed applications running on datacenter, enterprise and wide-area networks, the requirements on the underlying network fabric have become extremely stringent [51]. In particular, because the fabric interconnects applications' end-points, there is a push toward making it highly available.

A key factor that impacts fabric availability from the perspective of applications is failures. Even in the most well-managed networks, link/switch failures are common [19]. A variety of factors ranging from device crashes/reboots, cabling, buggy hardware/-firmware, power supply issues, etc., can conspire to constantly induce link/switch failures.

The fabric's behavior under failures critically determines its perceived availability. When a failure occurs in today's network fabrics, network forwarding attempts to reconverge to re-establish paths. When the network is still in an unconverged state, traffic destined to certain endpoints will have no valid route and will be dropped. This leads to a precipitous performance degradation for critical applications. Unfortunately, networks can remain in unconverged states for unreasonable amounts of time [30]. Local fast reroute (FRR) approaches [2, 3, 12, 33] can help mitigate packet losses when the network is unconverged by sending packets on alternate active ports. However, without any global state or advanced computation capabilities, FRR does not provide any guarantees about end-to-end paths under failures.

The other major consideration for networks is policy compliance. For example, Network Function Virtualization (NFV) [18, 39], a popular use-case, allows tenants and operators to specify middlebox chains that traffic between a set of endpoints must traverse for security and performance considerations. Because a non-trivial fraction of such middleboxes are now part of network fabrics [13, 16], the network is also tasked with ensuring correct middlebox traversal. As another example, operators may desire to employ various network load-balancing schemes—e.g., WCMP [50]—so that the network can effectively spread load across multiple available paths to avoid queuing and congestion drops. While operators can use various frameworks for policy compliance [6, 7, 15, 40, 46–48], ensuring that *policies always hold* when reacting to failures is something that no state-of-art approach achieves.

Thus, our primary goal is to design a network fabric which under failures can provide the illusion of being *always available* and *policy-compliant*. We define this as: *if, under a failure scenario, there exist active policy-compliant paths between a source-destination pair, then the fabric must route packets only through a policy-compliant path without inducing any drops.*

*Currently works at ByteDance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '21, October 11–12, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9084-2/21/10…$15.00
https://doi.org/10.1145/3482898.3483360

We observe that the main obstacle in realizing an "always available, policy-compliant" network under failures is that *recomputing new policy-compliant routes under failures is unreasonably slow*. Traditionally, recomputation is performed by a centralized or distributed control plane; in both cases, the computation is off the fast path of packet forwarding, and therefore slow. The data plane, which lies on the fast path, was only equipped to perform forwarding based on the control plane route computations. While fast reroute (FRR) mechanisms can sidestep the control plane limitation and quickly re-establish connectivity in the data plane, these mechanisms cannot provide policy-compliance guarantees, and do not always guarantee reachability.

Thus, to meet our goals, we argue for the data plane to take more responsibility under failures and perform route computation without waiting for the control plane. Our paper shows that, with technology available today, it is possible to realize such data plane-only routing that can instantaneously react to failures in a policy-compliant manner. The network uses either distributed protocols like OSPF/BGP or a centralized SDN controller to install the primary forwarding rules; our data-plane routing mechanism kicks in when the primary rule on a switch is invalid (similar to FRR). Our network mechanism, D2R[1], leverages recent programmable data planes to this end. Given a view of the network topology and current state of the links, D2R implements graph traversal algorithms—e.g., Breadth-first Search and Iterative-Deepening Depth-first Search—completely in the data plane; our implementation can compute paths to any destination at *near line rates*.

Because programmable switches today have limited processing stages, they may not be able to compute the route to the destination in one pass through the switch. We address this limitation using the recirculation capabilities of modern switches that allow packets to be fed back to the switch for additional processing. However, recirculations can cause latency and throughput degradation. Thus, we propose a hierarchical dataplane routing scheme that *significantly reduces* recirculations by splitting route computation across multiple switches. We evaluate our routing scheme on different topologies from the Internet Zoo dataset [29], and we are able to provide end-to-end routing in the data plane with less than 2 recirculations on average.

To achieve policy-compliance under failures, we augment our data plane traversal algorithms with support for different policies like middlebox traversals, local preferences and weighted load balancing. For quick rerouting under failures, we develop a policy plane which uses the end hosts to tag packets with policy headers, and our data plane's traversal algorithms use the policy to find compliant routes under failures. D2R shows the promise of enhanced data plane routing in current hardware switches. We envision D2R to complement current SDN systems by providing a mechanism for policy-compliant reroute under failures.

2 Routing under Failures

In this section, we present the challenges faced today w.r.t providing guarantees of connectivity under failures while complying with high-level policies. We examine the role of the control plane and data plane in different settings and make the case for general data plane routing under failures.

2.1 Control Plane Challenges

Network control planes primarily come in two flavors: using distributed routing protocols like OSPF/BGP, and centralized software-defined controllers. We discuss the limitations of both types of control planes in achieving always availability and policy-compliance.

Distributed control planes fall short. Many networks use distributed routing protocols that rely on routers exchanging protocol messages to convey changes in the network topology, for instance, when link failures occur. Each router uses these messages to recompute new forwarding tables to react to its perceived new state of the network. Until the information about failures propagates to all routers in the network, and the network has become quiescent, forwarding tables may not be consistent across routers. During this *convergence* period—which can last very long—severe packet losses occur when routes become unavailable [30].

Information about failures is passed via advertisements that are generated and processed by router software control planes. Francois et. al [17] study the behavior of IS-IS protocol convergence times for a 21 node topology geo-distributed in Europe and USA. They observe high convergence times of over *200-1000ms* depending on different control plane parameters—for instance, how the control plane updates the FIB can vastly change convergence times. In other words, switch/router control plane software design can further delay convergence, leading to higher loss rates.

Modern SDNs also fall short. Another approach to mitigate the impact of convergence is to leverage SDNs. In existing SDNs, a logically central controller manages a network of programmable switches. The controller detects failures, centrally computes forwarding rule changes, and pushes new rules to switches. However, this approach cannot be used to build a fabric which is always available. First, the controller must learn about the failure from network switches, which can incur high latency depending on the placement of the controller in the network, especially for wide-area networks (SDWAN) [22, 24]. Second, the controller may have to update the rules of multiple switches using complex update schedules so that intermediate network states do not lead to inconsistencies like packet loops and drops [35, 38, 41]. State-of-art SDN update mechanisms can take around *300ms to order of minutes* [38] to compute and install the update across a network. Further, He et. al [20] measure the latency for programming rules in OpenFlow switches: it can take *10-100*ms to add/modify/delete a single rule in the OpenFlow switch tables and such switch rule delays, mainly due to inefficient switch control plane software, make consistent updates even slower. Overall, even with SDNs, packets encountering failed links may be dropped for extended periods of time until failure notification and consistent update installation have completed.

Policy Compliance Challenges. In addition to ensuring availability under failures, we desire that packets always obey policies: routing around failures to reach a destination should not violate policies that pertain either to communicating with that destination or resource management.

In SDN, the centralized controller has to compute new sets of policy-compliant paths for different flows (identified by packet

[1]Pronounced "detour".

headers), which can be time-consuming. Determining the appropriate distributed control plane configurations where high-level policies are met (using techniques such as [7, 14, 48, 49]) is even harder. Thus, we cannot rely on the control plane to provide policy-compliance guarantees during failures.

2.2 Data Plane Challenges

To sidestep the control plane limitations, the data plane can employ various fast reroute (FRR) mechanisms, e.g., LFA-FRR [3], BGP-PIC [2], DFS/BFS on Openflow [8], PURR [12] and DDC [33] to forward packets on alternate paths when the primary path has failed. At a high level, these mechanisms are stateless (LFA-FRR, PURR) or maintain small state (DDC), and try different active ports on a switch which lead to the destination. FRR implementations are efficient and have very low state and processing overhead. However, the major drawback of these approaches is that they are built to work on switches which have limited computational abilities and cannot run complex algorithms. Hence, these approaches perform different distributed graph-search algorithms, where each switch performs simple operations and the network as a whole provides reachability guarantees. Packets may physically traverse to multiple switches in the network before it finds a path to reach the destination - which causes increased bandwidth utilization and higher chances of network congestion. Furthermore, due to limited visibility of the network topology, FRR mechanisms cannot always find an active policy-compliant path. With new switch architectures with enhanced processing capabilities [10], we can design FRR mechanisms which can avoid active exploration of packets in the network.

Consider LFA-FRR (Loop-Free Alternate Fast Reroute) [3] which can be used to protect 1-link failure by pre-computing and installing backup paths. However, LFA-FRR is not necessarily guaranteed to provide connectivity for certain topologies [42]. We performed an empirical analysis of using LFA-FRR for protection of every 1-link failure on topologies from the Internet Zoo dataset [29], and we observe that 2-20% source-destination pairs are disconnected until convergence under 2 and 3 link failure scenarios. Moreover, protection of multiple failures can lead to increased switch memory usage. PURR is a state-of-art FRR primitive for P4 programmable switches that can be used to implement general FRR sequences – a sequence of ports specified by some FRR mechanism [11]. Using PURR, the data plane can efficiently send traffic on the first active port in a sequence: FRR sequences can be used to provide connectivity even under multiple failures. PURR is a very efficient reroute mechanism-it uses less resources and does not incur recirculations. However, it can be difficult to encode different policies in the form of these FRR sequences.

Borokhovich et. al [8] propose local fast failover mechanisms, which are implemented on OpenFlow switches and provide provable connectivity guarantees under arbitrary link failures. They propose different flavors of BFS and DFS algorithms, which can be implemented using OpenFlow by tagging information on packet headers and using different tables such that packets physically explore the network in a depth-first or breadth-first fashion. Another state-of-art fast reroute mechanism is DDC [33] which uses fast link reversal algorithms in the data plane to provide provable connectivity guarantees. DDC maintains dynamic state on switches

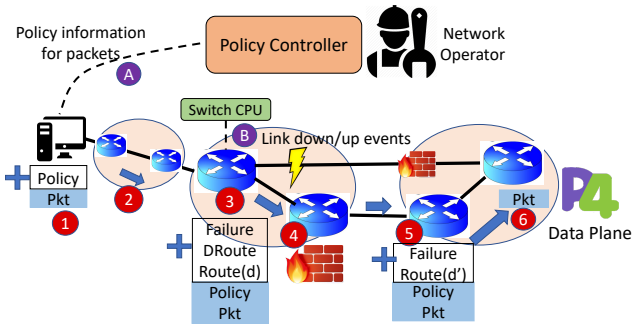


Figure 1: D2R architecture

and updates the state when packets are received. Dynamic state cannot be implemented on OpenFlow switches, and requires special register memory on P4 switches. Both these approaches do not have visibility of the entire network topology. They are difficult to extend to support network-wide policies.

3 D2R Architecture

D2R tackles the challenges mentioned in §2 and provides always availability and policy-compliance. We illustrate the D2R architecture in Figure 1. We partition the network topology into multiple domains (which are connected components) and construct a domain graph from inter-domain links. In D2R, the network control is divided into:

(1) **Policy Plane**: The centralized policy controller is used by operators to specify the *network topology* and *policy requirements* for different flows. The policy plane also sends the policy (A) to the end-hosts which are then included in the packet headers (1). Operators can choose enhanced guarantees for certain flows (encoded as a policy bit), while relying on local FRR for other flows to reduce the overhead on the data plane.
(2) **Data Plane**: The data plane uses programmable ASICs to run *hierarchical graph traversal algorithms*, atop the network topology encoded in the dataplane rules. When a failure occurs and the primary path fails, D2R is triggered and does not rely on the control plane or policy plane on the critical path (3-6). The data planes *encode link failure information in the packet header*, which is used for traversal (3). The data plane does not store global link failure state.
(3) **Control Plane**: D2R can co-exist with any distributed/SDN control plane (not shown in Fig. 1). The network control plane is responsible for the primary forwarding rules on the switch (2). The control plane is also responsible for monitoring link-up events in the switch (B).

We describe the flow of a packet in Figure 1: 1 The packet is tagged at the end-host with the policy header specified by the policy plane. In this case, we tag a firewall policy in the packet header. 2 When the packet enters the network, the primary forwarding rules are active, and hence, forwarding happens based on the converged network state, 3 The packet reaches a switch where the link is failed, and the primary forwarding rule is disabled (network has not converged to a new state yet). The D2R data plane first finds a domain path through the network (DRoute), and then computes a route through its domain (route(d)) to the firewall taking into

Policy	API	Description
Middlebox Chaining	addMboxChain(flow f, switch[] m1, switch[] m2 ...)	Chain of middlebox arrays where one middlebox is traversed in each array Can be coupled with BFS/IDDFS.
Next-hop Preference	addPreference(flow f, switch n1, switch n2)	From switch n1, prefer next hop n2 if n1 ⟶ n2 is active. Can be coupled only with IDDFS.
Weighted Cost Load Balancing	addWeightedLB(flow f, switch n, switch[] next, int[] weights)	At switch n, choose next-hop next[1] with probability $\frac{weights[1]}{\sum weights}$ ... Can be coupled only with IDDFS.

Table 1: D2R **Packet Policy Support**

account the failed links, and stores both routes as source routes in the packet. The failure information is also included in the packet. (4) The switch removes the firewall policy from the header, and then forwards to the next domain d'. (5) The switch computes a new route inside the domain d' to the destination (which is within the domain, so domain path is removed). (6) The packet is forwarded to the destination.

3.1 D2R Data Plane

Modern programmable switching ASICs let developers write complex packet processing pipelines that can run at very high speeds. For instance, the state-of-the-art Barefoot Tofino switch can process packets at an aggregate line rate of 6.5Tbps. Thus, in D2R, we move away from the conventional model where the data plane just forwards packets and the control plane runs sophisticated routing algorithms. Instead, the data plane runs hierarchical traversal algorithms like breadth-first search (BFS) and iterative-deepening depth-first search (IDDFS) to compute a route from the switch to the destination. Thus, each switch data plane has two components: a primary forwarding table (populated by the network control plane) and the D2R data plane routing mechanism. A packet is forwarded normally if the primary forwarding table has an *active valid* rule for the packet. D2R is triggered in the data plane if: (1) the primary outgoing port is disabled (failure), or (2) the primary outgoing port is the same as the input port (indicating transient forwarding loop). Thus, when a packet arrives at a switch ((3), (5)), D2R computes a route to the next domain/destination and stores the route in the packet header. D2R also updates one of the unused fields in the packet IP header to indicate that D2R has computed a route for the packet. For reachability guarantees before the network has converged, once D2R is triggered for a packet, subsequent switches ((4), (6)) will not use the primary forwarding rules (which could be inconsistent), and instead use the D2R mechanism for routing the packet. We describe our P4 [9] implementation of the data plane in §5.

Modern programmable ASICs can detect when a connected link is down and trigger a special packet indicating that the link/port is down ((C)). As soon as the failure is detected, the D2R data plane stores this information in a register. When a packet arrives, the data plane uses this updated local link-state and computes a route which does not use the failed link, avoiding any packet drops. This approach solves the problem faced by SDNs, in which failures cause the centralized controller to react and add new forwarding rules in a consistent manner. Our approach is more general than local FRR mechanisms, as the data plane can compute a valid end-to-end network path based on the current state of the links connected to the switch.

For correct routing in the network, we need to know the state of links in the entire network. However, we cannot wait for distributed link-state advertisements because this leads to convergence issues and packet losses. We eliminate routing convergence periods by extending the *Failure Carrying Packets* (FCP) protocol [31]. In FCP, each packet carries information about all the link failures it has encountered in its path ((3)). The switch data plane uses this information to find a route that avoids failed links without actually storing the current global link-failure state in the switch. FCP provides provable guarantees of connectivity under failures without the need for a distributed routing protocol. We describe the FCP protocol and implementation in §4.

3.2 D2R Control Plane

D2R can operate with any centralized/distributed network control plane which influences network forwarding behavior based on different considerations (service chaining, traffic engineering, access control etc.). When a failure occurs, the network control plane is responsible for installing the new primary forwarding rules in the switches. Between a failure and the subsequent convergence, the switch control plane plays no part in the critical path for end-to-end forwarding, and thus, it is not a bottleneck for always availability and policy-compliance.

The switch control plane also programs the D2R data plane with rules provided by the policy plane. These are *not forwarding rules* and instead they encode the network topology and any changes that occur to it in the long term, such adding/removing switches and links.

Some modern ASICs may not generate a packet for when the link has come back up. The switch control plane uses mechanisms like BFD [28] to monitor the status of links and notify the switch data plane of link up events ((B)).

3.3 D2R Policy Plane

D2R provides support for switch and network-wide policies under different failure scenarios. We restrict our support to *per-packet policies* in the data plane, i.e., computing a packet's route is independent from other packet routes. To support hyperproperties (a policy constraining the routing behavior of two or more flows), we would need to store routing state of different flows in the data plane, which would consume scarce switch memory resources.

Even for per-packet policies, we need to store the policy information for different flows. We could store the policy state in the switches, but if we needed to change the policies, we would need to reprogram switches, which can lead to down time (§2). Moreover, policy churn is significantly higher and can trigger frequent expensive network updates. Instead, we develop a policy plane which

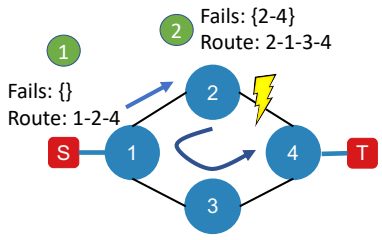


Figure 2: Example of FCP protocol in action when link 2-4 is down.

sends the policy information to end-hosts (A) which are responsible for adding the policy in the packet header (1). The data plane uses the policy header to generate policy-compliant paths when failures occur (3).

The policy plane can also request the current state of network links from switch control planes to generate new policies. Crucially, for the policies D2R supports, policy updates will not trigger reprogramming of the data plane. We describe D2R's policy support in Table 1 and the data plane implementation in §6.

4 Failure Carrying Packets

Failure Carrying Packets (FCP) [31] is a distributed routing paradigm designed to *eliminate* convergence periods altogether—a packet is guaranteed to reach the destination if a path to the destination exists in the network. FCP takes advantage of the fact that permanent network topology change (in terms of provisioning/deprovisioning links and switches) happens at the timescales of weeks/months and is well-planned. The only changes for which operators are not prepared for are links and routers failing and coming back up at smaller timescales [19]. Thus, each router has a consistent topology description which indicates all switches and adjacencies between them.

The intuition behind FCP is that if the switch knows the list of failed links in the network, it can successfully route a packet to the destination using the network topology and failure information. However, knowledge of all link failures will require a link-state advertisement protocol, which can lead to convergence issues. Instead, in FCP, each packet header carries information about all failed links it has encountered, and the switch simply uses the topology and failure information to route the packet to the destination. The packet on the route may again encounter a failed link to the next-hop, in that case, the failed links is added to the packet header and route is once again recomputed at the new router, and so on. We illustrate an example of the FCP protocol in Figure 2. Switch 1 computes the route $1 \rightarrow 2 \rightarrow 4$ (1) to the destination as it does not have any information about the failed $2-4$ link. When the packet reaches switch 2, the failure information in the packet is updated and switch 2 computes the new route to the destination $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$ (2). Switch 1 receives the packet once again, but it will not send the packet to 2 as the switch knows that $2-4$ is failed, and thus, sends it to 3 and so on. Algorithm 1 describes the FCP algorithm.

FCP is able to guarantee reachability if a path exists by the following intuition: at every switch in the network, the packet will monotonically increase the set of failed links in the packet[2]. Thus

[2]FCP does not consider link flapping—i.e., the packet encountered a failure and updated its header, but the link came back up before the packet reached the destination.

Algorithm 1 Failure Carrying Packet Protocol

```
1: procedure FCP(dst, router)
2:     pkt.failed_links ∪= router.failed_links
3:     path = ComputePath(topo - pkt.failed_links)
4:     if path == null then
5:         // No path to destination
6:         router.drop();
7:     else
8:         router.forward(pkt, path.next_hop)
```

eventually, the packet would get information about all failed links in the network, and any router would be able to route the packet to the destination if a path exists. The only failure state maintained by an FCP router is the failure state of links connected to the router. FCP learns about the state of remote links solely from the packet headers, and importantly, it does not store this information. Thus, FCP routers do not need to advertise failures unlike OSPF. Thus, while FCP can incur additional stretch, we can avoid the link-state flooding overhead during failures.

With programmable switch architectures, realizing a FCP-like protocol is more practical than when FCP was actually introduced. One of the major deployment challenges for FCP was changing the router hardware to support a new protocol header to incorporate information about link failures. With P4, we can easily define our custom protocol header and parsers, which can be efficiently run on hardware at line rates. We store the failure information in the header as a bit-vector where each bit represents the state of a particular link in the topology.

5 Hierarchical Data plane routing

Our hierarchical routing algorithm is inspired by OSPF's idea of dividing the network into areas to avoid large link-state databases on routers. We divide the network into n domains such that each domain is fully connected –i.e., there exists a path between each pair of switches in the domain. The choice of n determines the trade-off between the processing on a switch (smaller domain means lesser stages used for routing) and stretch of the path (more domains mean we find sub-optimal paths due to limited visibility). We construct a domain graph based on the domain adjacencies—i.e., if there is a switch n_1 of domain d_1 connected to switch n_2 of domain d_2, we add an edge between d_1 and d_2 in the domain graph. Hierarchical routing works as follows: (1) The source switch computes the domain path to the destination domain and stores the path in the packet header. (2) The source switch then computes the intra-domain network path to a switch that belongs to the next domain in the domain path, and sends the packet to that domain, and so on till we reach the destination domain. (3) The switch in the destination domain finds a path to the destination. In summary, instead of finding the complete network path in a single switch, we split the computation across multiple domains, and the first switch is also responsible for finding a path in the domain graph. D2R can use either IDDFS or BFS (this choice is made by the policy plane as certain policies are only compatible with IDDFS). In this section, we first present a primer on programmable switches and P4, the state-of-art language used to program these switches. We

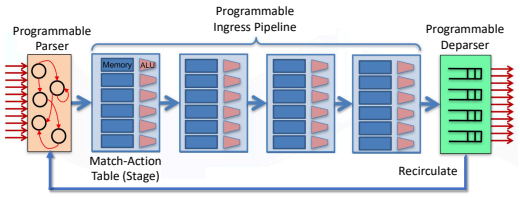


Figure 3: Ingress pipeline in programmable switches

then present the non-hierarchical flavor of two graph traversal algorithms we implement in D2R: breadth-first search (BFS) in §5.2 and iterative-deepening depth-first search (IDDFS) in §5.3. We then present how we can implement our hierarchical routing logic in §5.4.

5.1 Programmable Switches and P4

Modern programmable switching ASICs [10] contains three main components: the ingress pipeline, the traffic manager, and the egress pipeline. A switch can have multiple ingress and egress pipelines serving multiple ingress and egress ports. Packet processing is performed primarily at the ingress pipelines (Figure 3) which comprises of three programmable components: a parser, a match-action pipeline, and a deparser. To support complex packet processing, each pipeline has multiple stages which process packets in a sequential fashion. Each stage contains dedicated resources (e.g., match-action tables and registers) to process packets at high rates. For instance, the state-of-the-art Barefoot Tofino switch can process packets at an aggregate line rate of 6.5Tbps.

Packet processing can be abstracted as a control flow graph of match-action tables, where each table matches a set of header fields, and performs actions based on the match results. While processing a packet, the stages of the ASIC share the packet header and metadata fields (can be thought of as global memory), and stages can pass information in the pipeline by modifying these headers. The number of stages in programmable switches is limited, and the packet processing logic may not finish at the pipeline. In such scenarios, the packet can be *recirculated* back into the ingress pipeline with updated headers for further processing. Recirculating a packet multiple times consumes switch bandwidth resources (ports are set up in loopback mode for recirculations and cannot be used for physical links) and results in increased latency. Note that we will only incur these recirculations for packets whose primary forwarding rule has been affected by a failure which triggers D2R - hence, packets are not recirculated if either primary forwarding path is unaffected. However, to avoid overhead due to D2R, our data plane algorithms must reduce the number of recirculations required for packet processing.

P4 [9] is the most widely used domain-specific language to program these ASICs. While P4 is a programming language, it closely mimics the architecture of programmable ASICs—i.e., we cannot express any general algorithm as a P4 program. Thus, we need to take into account the P4 semantics for designing our graph traversal algorithms and express steps of the routing algorithms as match-action tables.

Algorithm 2 Stack-based Breadth First Search

```
1:  procedure BFS(src, dst)
2:      Initialize array of Stacks[MaxDepth]
3:      depth = 0
4:      curr = src
5:      while curr != dst and Stack array is not empty do
6:          for next in Neighbors(curr) do
7:              if (curr, next) is not visited or failed then
8:                  // Add valid neighbor to stack of depth + 1
9:                  Stack[depth + 1].push(next)
10:                 Mark all incoming edges to next as visited
11:                 Parent[next] = curr
12:                 goto While
13:         if Stack[depth] is not empty then
14:             // Explore next switch at current depth
15:             curr = Stack[depth].pop();
16:         else
17:             // Explored all switches at current level. Move to depth + 1
18:             depth++
19:             curr = Stack[depth].pop();
20:     Traverse Parent map from dst→src to compute path
```

5.2 Breadth First Search

We now present the algorithm and P4 implementation for performing breadth-first search (BFS) in the network using the FCP header such that the computed route does not traverse any of the failed links in the FCP header. BFS has the advantage of finding paths with the least number of hops. Traditionally, BFS explores the switches of the graph using a first-in-first-out (FIFO) queue. However, since currently P4 only supports stack data structures, we implement a modified BFS algorithm in P4 which uses only stacks and preserves the following invariant: *a switch at a lower depth (number of hops from the source) is explored before any switch at a higher depth.* The only difference from a queue-based implementation will be the relative ordering of explored switches at each depth. We present our stack-based BFS algorithm in Algorithm 2 and in the rest of the section.

P4 implementation. In programmable switches, the amount of memory to store packet headers and metadata is limited. Since the BFS stacks need to be processed by every stage, we need to store it as a header field[3], thus, we must limit the number of used stacks. Our BFS algorithm uses two stacks for odd (Stack[1]) and even depth (Stack[0]) switches, respectively— when we are exploring switches of odd depth d, we push the neighbors at depth $d + 1$ in Stack[1] and vice-versa, eliminating the need of more than two stacks. We now break down how we translate Algorithm 2 to P4. The building block of our BFS algorithm is the *bfs* P4 match-action table.

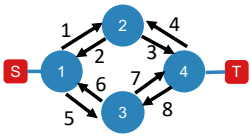
Initialization. We initialize `curr` to the switch that is computing a path to the destination. `visited` is a bitvector whose size is equal to the number of bidirectional links. For each $link_i$, `visited[i]`[4] is set to 1 if $link_i$ has been visited or has failed, and to 0 otherwise. We set all failed links obtained from the FCP header to 1. Consider Figure 4, if $1 \rightarrow 2$ has failed, then we set the 1^{st} and 2^{nd} bits of `visited`—0000 0011. We also set all incoming links to `curr` to 1, so that BFS does not visit `curr` later in the algorithm. For

[3]We will not emit these stacks in the deparser as they are not required for correct forwarding in the network.
[4]The indices start from 1 from the rightmost bit of the vector.

Algorithm 3 Iterative Deepening Depth First Search

```
1:  procedure DFS(src, dst)
2:      Initialize empty Stack
3:      curr = src
4:      len = 0
5:      max_len = 4
6:      while curr != dst do
7:          if len < max_len then
8:              for next in Neighbors(curr) do
9:                  if (curr, next) is not failed or visited then
10:                     // Go to valid neighbor
11:                     Mark all incoming edges to next as visited
12:                     Stack.push(curr)
13:                     Parent[next] = curr
14:                     curr = next
15:                     len = len - 1
16:                     goto While
17:         // No valid neighbor. Backtrack
18:         curr = Stack.pop()
19:         len = len - 1
20:     else
21:         // current length exceeds max length. Backtrack
22:         curr = Stack.pop()
23:         len = len - 1
24:     if curr == NULL and Stack is empty then
25:         // Explored all switches within max_len distance
26:         // Increase max_len exponentially
27:         max_len = max_len × 2
28:         curr = src
29:         Reset visited state
30:     Traverse Parent map from dst→src to compute path
```

```
table bfs { key={
    hdr.curr : exact;
    hdr.visited : ternary;
    hdr.stack: exact;
  } actions =
  {push_neighbor; pop_stack; change_stack;}
```



curr	visited_vec	Action parameters
1	*******0	n = 2, n_visited |= 00001001
1	***0****	n = 3, n_visited |= 10010000
2	*****0**	n = 4, n_visited |= 01000100
3	*0******	n = 4, n_visited |= 01000100

Figure 4: Example topology of 4 switches and the subset of BFS table rules.

our 2-stack implementation, we denote switches at odd depth with `stack` = 1, and switches at even depth with `stack` = 0.

Let m be a switch at odd depth. BFS explores a neighbor n which is unvisited and connected (line 9) by an active link and puts in Stack[1] (line 9). To map our algorithm into the P4 programming model, we translate the if condition to a table match rule, and the code executed based on the if condition as one of the table actions.

If the link ID of $m \rightarrow n$ is id, then we will only explore n from m if `visited[id]` = 0. Consider the example in Figure 4. If m = 1, we will only explore $1 \rightarrow 2$ if `visited[1] = 0`; P4 supports ternary match kind for bitvectors where we can specify exact values (0/1) or wildcard for each bit; we use the ternary match to check the id^{th} bit in `visited`. Thus, the match fields for exploring the edge $m \rightarrow n$ would be as follows (depending on if m is at odd or even distance from source):

```
match = curr: m=1, visited: *******0, stack:0
       curr: m=1, visited: *******0, stack:1
action push_neighbor(n, n_visited) {
    Stack[~hdr.stack].push(n);
    hdr.visited = hdr.visited|n_visited;}
```

If the above match succeeds, we need to push n = 2 onto Stack[1]. We also set all the bits corresponding to incoming edges to n as 1 in `visited`; thus, BFS will not explore n again. `push_neighbor` action implements the lines 9-10 [5].

Figure 4 shows the action parameters when we explore the edge $1 \rightarrow 2$. Once, all neighbors of m are explored, the BFS algorithm will pop the next element from the stack of the current depth (odd or even) and repeat the process of exploring the neighbors (lines 13-15). To check if all neighbors of m have been explored, we again use the ternary match to check if all bits corresponding to outgoing links of m are 1. If so, we update `curr` to the top switch of the stack. For example, if m = 1, the links with ID 1 and 5 must be explored:

```
match = curr: m=1, visited:***1***1, stack:0
       curr: m=1, visited:***1***1, stack:1
action pop_stack() {
    hdr.curr = Stack[hdr.stack].pop();}
```

Finally, once we have explored all switches in the stack, we need to proceed to the switches at the next level. To match for this condition, we place a special switch "0" at the bottom of stack and swap stacks when once `curr` = 0. After switching stacks, we pop the top element of the new stack to start exploring its neighbors.

```
match = curr:0, visited:********, stack:0
       curr:0, visited:********, stack:1}
action change_stack() {
    hdr.stack = ~ hdr.stack;
    hdr.curr = Stack[hdr.stack].pop();}
```

Ingress implementation. According to the P4 semantics, only one match-action rule will be triggered per table application. The match condition depends on the current packet headers, priorities and ordering of rules in the table. In the ingress pipeline, when a table is applied, the switch will execute the action code corresponding to the matched rule. Thus, a single `bfs` table application cannot perform the entire traversal. We apply multiple `bfs` tables to perform BFS from the source till `curr` is the destination switch.

The `bfs` tables cannot be placed in the same stage due to Read-After-Write (RAW) dependencies [27]. Current off-the-shelf switches only have a bounded number of stages (~10) in the ingress pipeline. Therefore, our BFS algorithm may not reach the destination in

[5] P4 targets may not support specifying header fields as indices— we define two action push_neighbor_0 and push_neighbor_1 to push onto Stack[0] and Stack[1] respectively. We elide these details for simplicity.

those stages. To overcome the limitation of bounded number of stages, we can repeatedly *recirculate* the packet back into the ingress pipeline with the headers at the end of the pipeline. This effectively resumes the BFS algorithm, and we keep applying the `bfs` table till we find the destination in the algorithm. Note that the amount of recirculations is affected by the number of stages— a switch with more stages will incur lesser recirculations. We implement a source routing flavor of BFS— the route is stored in the packet headers and downstream switches can use the source route and avoid unnecessary route recomputations (thus, avoid extra recirculations). The scalability of our system is tied inherently to the number of stages in the switch. However, given that recirculation is local to the switch, D2R with multiple recirculations (in the order of μs) would be able to react much faster than mechanisms which require communication across network links (which are of order of *ms*). Also note that D2R can be run in conjunction with other data plane functionalities. Modern programmable switches contain support to run multiple pipelines in parallel, thus, switches can run other functions in parallel to the D2R tables which is only responsible for computing paths and does not manipulate other packet headers.

5.3 Iterative Deepening Depth First Search

Another form of graph traversal is Depth-first Search (DFS). However, without bounds on the path length, DFS can produce very long paths compared to BFS. This is not ideal, especially in wide-area settings. We implement a variant of DFS called Iterative Deepening DFS (IDDFS), which explores switches in a manner similar to DFS while imposing bounds on the length of the discovered paths, and iteratively increases the bound when needed. We present our IDDFS algorithm in Algorithm 3.

IDDFS works similarly to DFS with one major modification: we keep track of the length of the current path from src (`len`) and will not explore neighbors if the length of the path exceeds the max length. Thus, IDDFS provides bounds on the path length and will eventually find a path if one exists within the bound. If a path within the bound does not exist, we perform DFS with an increased bound. IDDFS is linear in complexity. In the worst case, it explores $2N$ switches.

P4 Implementation. Similar to BFS, we create a P4 table which acts as the building block of our IDDFS algorithm.

```
table iddfs {
  key = {
    hdr.curr : exact;
    hdr.visited : ternary;
    hdr.len: exact;
    hdr.max_len: exact;
  } actions =
  {goto_neighbor; backtrack; increase_length;}
  default action = backtrack();
```

Similar to BFS, we add table rules to check if certain edges are visited/failed (using ternary match) and explore neighbors. Backtracking occurs when we have no neighbor to visit from a switch. Finally, we increase the maximum path length when the stack is empty— i.e., when we have explored all switches at the specified maximum length but did not reach the destination. The P4 Implementation details are in the appendix of the supplementary material.

As with BFS, each invocation of the `iddfs` table can lead to one action execution. Thus, we add n tables staged one after the other (due to the RAW dependency). At the last stage, if we have not found the destination, we recirculate the packet again. Similar to BFS, we implement source routing for IDDFS. IDDFS *requires* source routing for *correctness* purposes as it does not compute the shortest path to the destination. Consider the topology in Figure 4. Switch 1 uses IDDFS and computes the route $1 \rightarrow 2 \rightarrow 4$ (but does not store it in the packet) and sends to switch 2. Switch 2 now performs IDDFS to compute route $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$, and sends it back to 1, and thus, the packet will keep oscillating. Oscillation is circumvented by source routes: switch 2 will simply use the source route to send to 4.

5.4 Hierarchical Routing

We extend our graph algorithms to perform a traversal over the domain graph and store the domain path in the header, which is then used by the switches to find a path through each of the domains. We use the BFS/IDDFS tables defined in §5 for finding both the domain path and the network path, and differentiate between the two modes using a header field in the table match conditions: `hdr.hierarchy = 1` means we are finding a domain path, and `hdr.hierarchy = 0` means we are finding a path inside the domain. We implement the switching logic between inter- and intra-domain routing in our ingress table actions (details omitted for brevity).

Routing Inside Domains. A switch has to find a route to one of the switches in the next domain. We modify the topology of each domain to add a special switch for each neighboring domain. We take all inter-domain links and connect them to the special domain switch. We illustrate this augmentation in Figure 5. Thus, to find a path to the next domain d, we set `hdr.destination = d` and perform BFS/IDDFS on the modified topology—thus, finding a valid path to the next domain. Consider a packet from S to T in Figure 5. Switch 1 will first compute the domain path to T which is $128 \rightarrow 129$. Then, it will perform intra-domain BFS/IDDFS to 129 in the augmented intra-domain graph and will reach either switch 4 or 5 (based on if route is computed through 2 or 3, respectively). Switch 2 and 3 will have forwarding rules to send the packet to 4 and 5, respectively. Once the packet has reached a switch in domain 129, the switch can perform intra-domain routing to reach the destination.

Hierarchical Routing under failures. We modify the FCP failure vectors to account for inter-domain link failures. Consider the example in Figure 5: the domain link 128 - 129 can be marked as failed only if both $2-4$ and $3-5$ links have failed. Thus, we can create a mapping of the network failvector to domain failure vector (implemented using a match-action table)—the domain failure vector can be then used to perform traversal on the domain graph. However, unlike normal FCP routing, hierarchical routing does not provide strict guarantees of reachability: if a domain becomes internally disconnected due to multiple failures, we may not find a route to the destination even if one exists. In such a scenario, we could choose to the send the traffic back to the previous domain and mark the domain link as down, so the previous domain will try to find a new domain path through the network. Choosing a domain assignment which minimizes the occurrence of domain

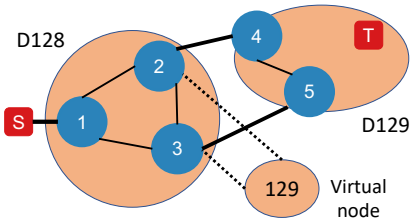
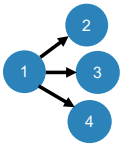


Figure 5: Example of hierarchical routing. For domain 128, we add virtual switch 129 and add links corresponding to $2-4$ and $3-5$.



curr	visited_vec	pref	Action
1	*******0	**	next = 2
1	******0*	**	next = 3
1	*****0**	**	next = 4
1	******0*	1*	next = 3
1	*****0**	11	next = 4

Figure 6: Preference values for a switch with 3 next hops

disconnections along the lines of prior work [48] would help provide stronger guarantees under failures. The diameter of a domain impacts the length of the paths (and subsequently the number of stages required). Choosing a good domain assignment is subject for future work.

6 Policy Implementation

The D2R data plane can find compliant routes for the packet policies listed in Table 1, even under failures. The operator specifies the policies to the policy plane using the API, and the policy plane specifies the policy information which must be sent on the packet, which is used to guide the traversal. In this section, we present the modifications to our IDDFS implementation to support policies. D2R has the following failure semantics: if there exists a policy-compliant route in the network, the data plane algorithm will find it. One of the biggest advantages of D2R's policy support is the encoding of policy in the packet which ensures that any change in policy does not require reprogramming of the network - the data plane is intelligent to compute new paths to satisfy the new policy in the packet.

6.1 Middlebox Chaining

With the emergence of NFV [18, 39], operators can place middleboxes at different locations in the network to perform different network functions—e.g., firewall, intrusion detection, traffic optimizers etc., With the middlebox chaining policy, operators can specify a chain of middleboxes m1 $\rightarrow$ m2 ... and the data plane must compute a path from src to m1, then to m2 ... and then to the destination. The middleboxes and destination are encoded in the packet header, and the data plane sets `hdr.dst` = m1, so then IDDFS will find a route to m1. Once, the path is found to m1, we set `hdr.destination` = m2 and restart traversal from m1, and so on until the switch computes a route to the destination. The other switches can use the computed route for forwarding. Under failures, a switch will be able to compute a new route through the middleboxes. In a non-hierarchical setting, each switch has visibility over the entire network, however, in the hierarchical case, a switch can only route to middleboxes within its domain. Each switch stores the switch-domain mapping, and a switch will first find a domain path which traverses through the domains which contain the middleboxes. Inside a domain, a switch can find a intra-domain path traversing the middleboxes in the domain.

We also have support for specifying middlebox replicas. With support for disjunctions in P4 conditional statements, we can end IDDFS when `hdr.curr` is equal to any of the middlebox instances, which ensures that the switch can dynamically pick one of the reachable middleboxes on-the-fly. While FRR schemes like DDC can be extended with IP-in-IP encapsulation schemes to support simple waypoint routing, they will not be able to dynamically pick one of the middlebox replicas. We add multiple fields in the header to store the replicas and modify our ingress pipeline as follows:

```
control ingress {
  if (curr != dst[1] && curr != dst[2]...)
    // apply iddfs
  else
    // switch to next middleboxes/destination
    // or forward to next-hop
```

Note that enforcing the middlebox policy will not incur any additional rules or per-flow state; the policy in the packet header will specify the middlebox chain and replicas, which will be read by the data plane.

6.2 Next-hop Preferences

Operators may impose the most preferred path among multiple paths available to a destination, so that the fabric prefers or avoids using certain paths for cost or performance reasons. Preferences can be used by the operator to send a particular class of traffic through a geographical domain which has higher bandwidth or is less prone to malicious entities. D2R supports next-hop preferences (akin to BGP local preferences), which can be used to specify at switch n the best next-hop b for the packet. To enforce this policy in the data plane, we need to ensure that when our traversal reaches n, it must choose $n \rightarrow b$ if the link is active and routes to the destination. For next-hop preference, we use the IDDFS traversal to find a route. In IDDFS, the hop which is explored first is the most preferred hop (as IDDFS will move to b and so on till it finds the route to destination), thus, we need to enforce that the rule $n \rightarrow b$ is matched first in IDDFS. We cannot use rule priorities as they will require control plane intervention for different policies.

We add a new longest prefix match (lpm) field to the `iddfs` table: `hdr.pref`. For each switch and next-hop, the policy plane decides the `pref` value to guide IDDFS towards the most preferred hop. We illustrate the preferences using an example in Figure 6. Suppose the policy specifies that 4 is the most preferred hop from 1, for which the `pref` value is set to 11. By virtue of the lpm match, the 5^{th} rule will be the most preferred rule and IDDFS will explore 4. Similarly, if we set `pref` = 10, the switch will match to the 4^{th} rule and switch 3 will be the most preferred route. Finally, if we set `pref` = 00, all $1^{st} - 3^{rd}$ rules are valid matches with equal length prefixes (**). According to the P4 switch semantics, the first rule will be matched, and IDDFS will explore switch 2. The policy plane is responsible

for specifying the right preference value in the packet depending on the policy, and the data plane will explore the appropriate hop if it is active. We do not support backup preferences in the data plane (prefer b1, then b2 etc.). However, if the preferred link is down, we ensure we pick an active route (to ensure high availability).

6.3 Flexible Weighted Load-Balancing

One of the key responsibilities of network routing is load-balancing—sending different flows on different paths to manage network capacity. D2R supports flexible WCMP [50] in the data plane—i.e., the packet will carry the WCMP weights for a switch, and the switch's data plane will find a route by picking a next-hop with probability calculated by the weights specified in the packet. The data plane logic does not depend on any particular set of weights. Thus, we can simply change weights in the packet and the data plane would perform load-balancing according to the new weights. In current networks, the control plane needs to add a set of rules based on fixed WCMP weights—if one needs to change weights, the control plane needs to modify the data plane rules.

We illustrate how D2R avoids this problem. Consider the switch in Figure 6. Assume the policy in the packet specifies load-balancing weights as 1:2:1. We use preferences presented in §6.2 to load balance flows according to the weights in the packet. The data plane should set `hdr.pref = 00` with probability 1/4 for switch 2, `10` with probability 2/4 for switch 3, and finally, `11` with probability 1/2 for switch 4. Thus, flows will be load-balanced at switch 1 with weights 1:2:1. P4 switches have support for generating hashes from the packet header fields, which D2R uses to decide the next-hop preference in a probabilistic manner. To support flexible WCMP, we use Boolean operations in a preprocessing table to map the random hash to a preference value based on the input weights. In the face of failures, we prefer a next-hop from the active next-hops with the same relative weights. For example, if the policy for switch 1 in Figure 6 is 1:2:1 and link $1 \rightarrow 3$ is down, the links $1 \rightarrow 2$ and $1 \rightarrow 4$ will be preferred in a 1:1 ratio. We do flow-level load balancing as our WCMP hash function uses the packet header fields, so packets of the same flow will be sent to the same next hop. This ensures packets in a flow are not reordered. For brevity, we elide the P4 implementation details.

Policy Support Limitations. We currently do not support next-hop preferences and weighted load balancing policies with BFS traversal. BFS explores multiple routes simultaneously, so choosing one of the BFS routes which comply with the policy requires more complicated processing in the tables and increased header state, thus, inflating the number of stages required to find the path (thus, more recirculations). BFS works in conjunction with middlebox policies.

7 Implementation and Evaluation

The implementation of the D2R data plane consists of ~2000 lines of $P4_{16}$ which can be run on the P4 software behavioral model [1] that emulates the behavior of programmable switch architectures. The policy plane (~3000 lines of Python) uses the topology specification to generate the D2R rules for each P4 switch in the topology. The policy plane hands off these rules to the switch control plane which uses the switch APIs to install the table rules in the software bmv2 and hardware switch.

Both of our hierarchical graph traversal algorithms (denoted as H-BFS and H-IDDFS) use 10 stages (configurable parameter) to run the tables outlined in §5. In our experiments, we store 8 hops in the header. We evaluate the effectiveness of routing using D2R using the Internet Zoo topologies [29] (5-52 switches, 10-126 links) in terms of recirculation overhead and path stretch. We split the network into random contiguous domains, each domain containing 5-10 switches. We evaluate under different failures scenarios varying between 1 and 3 links.

In hardware, switches generate a packet to indicate a link is down and the delay between the actual link failure and packet generation is a few microseconds. At 10Gbps, the data loss occurring between actual failure event and the data plane reacting to the failure will be in the order of kilobits, i.e, 1-2 packets (thus, nearly zero drops). For the rest of the section, we assume that failure detection is instantaneous.

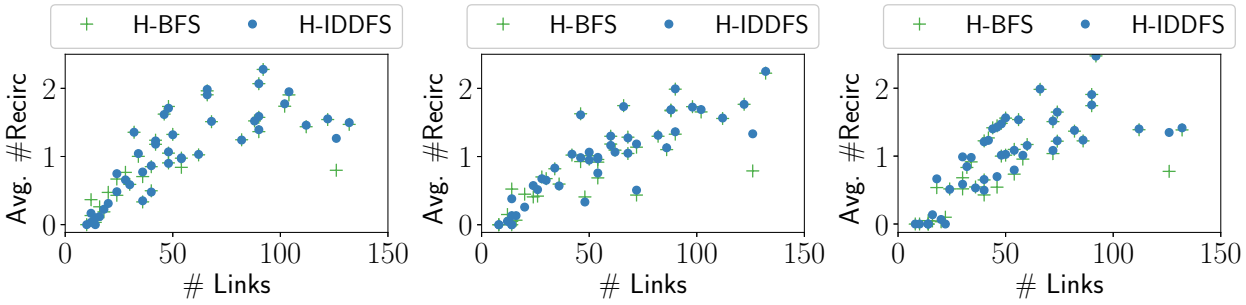
7.1 Routing Effectiveness

We evaluate D2R's ability to find routes using H-BFS and H-IDDFS, and measure the number of recirculations and path stretch incurred by both techniques. For these experiments, we generate packets for all pairs of endpoints in the network and emulate the data plane behavior using bmv2. We simulate the network by analyzing the output packet from bmv2 and "forwarding" it to the next switch. To evaluate D2R under failures, we generate 20 failure scenarios for each number of failed links $k = \{1, 2, 3\}$, and observe the routing behavior for all-to-all traffic. We do not include policies for these experiments as they do not affect recirculations or path stretch intrinsically: middlebox policies will inflate the path and recirculations due to longer path traversal, while preferences and load-balancing have no effect on recirculations.

Packets undergo route computations at multiple switches (at the start of a new domain or due to failures), thus, we report the total network recirculation in Figure 7. We define the path stretch as the ratio of the actual path taken by the packet in the D2R network compared to the shortest active path in the network (computed by an oracle using BFS). We report the stretch for the networks in Figure 8.

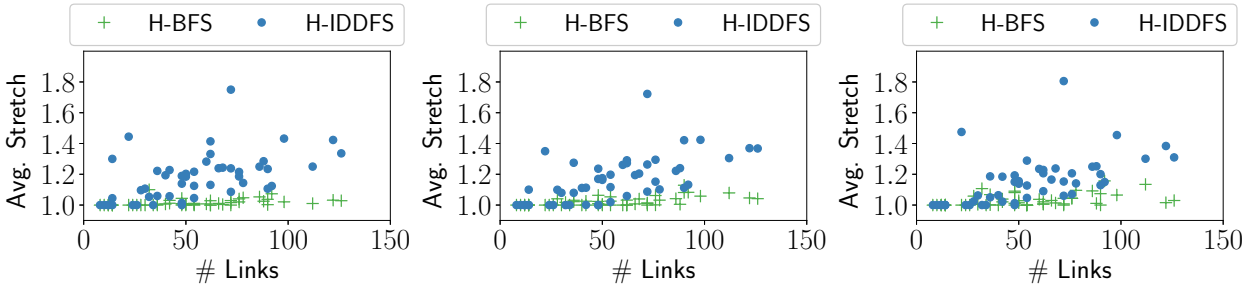
D2R can find routes using few recirculations (average < 2.5) for the different networks, and we observe more recirculations as the network size increases. This increase is expected; we need more processing to explore the switches and links (split across domains) to find a route. In the presence of failures ($k > 0$), the FCP algorithm kicks in and D2R needs to recompute paths on multiple switches as packets learn about new link failures. However, this does not significant affect recirculation, and we find average recirculation to be consistent across different failure scenarios. This is mainly because, failure may only affect a small subset of traffic (depending on the topology) and traffic will not encounter all failed links. We also observe that both H-BFS and H-IDDFS algorithms incur similar recirculations.

We observe a similar trend with path stretch. H-BFS has low path stretch (<1.1) and is effective in finding the shortest path, even across domains. For our experiments, we start H-IDDFS with maximum length as 4 and increase step size by a factor of 2. Thus, H-IDDFS incurs a higher stretch as it does not always find the



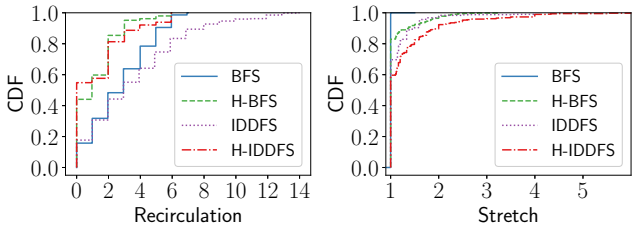
(a) k=1 (b) k=2 (c) k=3

Figure 7: Average network # recirculations for varying networks under different k-link failure scenarios.



(a) k=1 (b) k=2 (c) k=3

Figure 8: Average stretch (ratio of length of path taken vs. shortest path) for varying networks under different k-link failure scenarios.



(a) USA-Recirc (b) USA-Stretch

Figure 9: Recirculation and stretch CDF for BFS and IDDFS with and without hierarchy for NetworkUsa (78 links). BFS and IDDFS represents non-hierarchical routing.

shortest paths. H-IDDFS does not incur a very high stretch (<1.6) for most topologies.

Packet header overhead. In our hierarchical routing scheme, each packet only carries failure status for a subset of links. For the topologies we considered, this failure information can be stored in a 64-bit header. The total D2R packet header overhead (for storing source routes, policies etc.) is ~300 bits, which is small compared to the total packet sizes (2.5%), and is only incurred during periods of convergence.

7.2 Benefits of Hierarchy

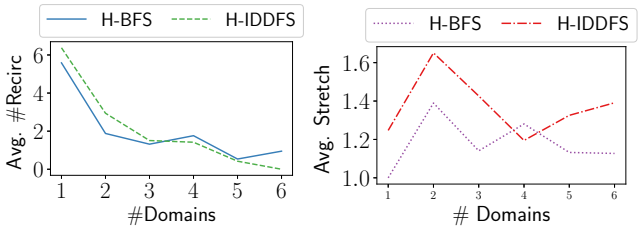
We also evaluate the benefits of using hierarchical routing compared to non-hierarchical routing algorithms. For clarity of exposition, we focus on one topology from Internet Zoo - NetworkUsa with 35 switches and 78 links. We partition the network randomly into 3 domains of roughly equal size. We consider all pairs of switches as endpoints and plot the cumulative distribution of the total network recirculation in Figure 9(a) and path stretch in Figure 9(b) for all strategies.

Hierarchical routing results in a significant reduction in recirculation for both BFS and IDDFS, we are able to bring the maximum recirculation to 6 in the hierarchical scheme compared to 14 in non-hierarchical scheme. Remarkably, hierarchical routing achieves 0 recirculations for 50-60% of endpoints, reducing throughput and latency degradation. Finally, most of the traffic suffers low stretch even with hierarchical routing (80% traffic have a stretch <2 with hierarchies).

We conclude this section by evaluating the effect that varying the number of domains has on the number of recirculations and path stretch (Figure 10). As the number of domains increases, the average recirculations decreases. The effect on stretch with changing domains is harder to analyze, as the stretch depends on the topology structure and how the domains are assigned. However, we note that generally higher stretch is incurred when the number of domains is > 1 because a switch computes routes on a partial topology. In summary, hierarchical BFS and IDDFS are able to significant reduce recirculations for the endpoints without a significant increase in average path stretch (1.2-1.6×).

7.3 D2R in the Real World

We run D2R on a Stordis BF606X switch which can run P4 Tofino programs. The first aspect of running D2R on hardware is compiling to Tofino. We use Barefoot P4 Studio [5] to compile a version of D2R that adheres to the resource constraints of the switch. Since



(a) # Recirculations (b) Stretch

Figure 10: Average recirculation and stretch for NetworkUSA (35 switches, 78 links) with varying number of domains.

we only possess a single hardware switch, we could not perform an end-to-end routing demonstration using D2R. To understand the viability of D2R, we study the effects of recirculation on Tofino. By configuring adequate ports in loopback for recirculation, we are able to run D2R on the switch and can perform 2 recirculations with minimal degradation in throughput and additional latency in the order of microseconds between two hosts connected to the switch[6]. Note that recirculation decreases the usable bandwidth of the switch, but routers in the topologies we considered have small average degrees (2-4). Thus, the switch has a lot of unused ports (for e.g., 65) whose bandwidth can be used for recirculation without affecting the active links' line-rate processing. While the available switch bandwidth reduces, we would still be able to process packets on links for these topologies at line rates.

End-to-end connectivity using Mininet. We demonstrate end-to-end routing of D2R using an emulated Mininet [4] network of four P4 switches and two hosts (Figure 4). The P4 switches run the P4 software behavioral model [1]. The bmv2 CLI is used to program the switch rules for each switch for routing and forwarding packets in the network. We send UDP traffic from S to T with the D2R headers as payload. We disable link $1-2$ (using a link failure status register in the switch). When $1-2$ has failed, 1 successfully finds an alternate path $1 \rightarrow 3 \rightarrow 4$ using IDDFS without any packet drops (assuming failure detection is instantaneous). We also verify that switch 3 does not compute the paths, instead uses the route installed in the packet header.

8 Related Work

PURR [12] is an efficient local fast reroute primitive to implement general FRR sequences. The switch will try to send the packet on the first active port in a sequence. To re-establish connectivity, operators can implement multiple mechanisms that use the FRR primitive to explore paths in the network using different strategies—e.g., Rotor-Router, DFS, BFS [44], and F10 [34]. While PURR does not incur recirculations and uses less resources, we argue that D2R can provide enhanced routing capabilities in the data plane, e.g., by supporting complex policies.

DDC [33] is another state-of-art data plane mechanism that can provide provable connectivity guarantees using link-reversal algorithms. DDC constructs a directed acyclic graph (DAG) for each destination which marks the traffic flow under no failures. Links are either marked as incoming or outgoing, and when a switch receives a packet, it will try to send to an outgoing link. If the packet is received back from the link (meaning there wasn't a path to the destination), DDC reverses the direction of the link and tries another link. By performing a series of link reversals, the packet eventually reaches the destination. Each switch only stores the direction of it's local links and updates the direction based on where packets are received. Thus, DDC is very lightweight compared to D2R but requires maintaining dynamic state of the links on the switch.

Molero et. al [37] propose a path vector protocol using programmable switches, and offloading key control plane functionalities to the data plane, in the same vein as our vision. However, a distributed path vector protocol, even one accelerated by hardware, will suffer losses during routing convergence periods. Blink [21] is a state-of-art data-driven data plane solution for connectivity recovery. Blink analyzes TCP-induced signals to detect remote link failures that disrupt end-to-end connectivity. Once Blink has detected a remote link failure, it uses a very simple data-driven fast reroute mechanism: it probes all next hops for availability and chooses a working one. D2R could be potentially used as Blink's reroute mechanism.

Contra [23] is a general and programmable system that uses P4 switches to achieve performance-aware routing. Operators can specify routing constraints and performance objectives in a high-level language and these constraints are translated to P4 programs to run on switches. Contra generates periodic probes that traverse policy-compliant paths, and switches run a form of specialized distance-vector protocol to decide the best path for traffic based on the performance metrics and routing constraints updated by the probes. Contra suffers from convergence problems under failures - the switches will require re-advertisements to learn about failures during which packets can be dropped. D2R is orthogonal to Contra, and can be used in conjunction with Contra to provide guarantees till the Contra protocol converges to a policy-compliant path.

Finally, one of the major avenues of research orthogonal to work is leveraging programmable data planes to perform various in-network computing tasks efficiently: key-value stores [26], scale-free coordination for distributed systems [25], stateful load balancers [36], network ordering for consensus [32], heavy hitter detection [45], and distributed aggregation for machine learning [43]. We could potentially run D2R and these applications in parallel in the same data plane with D2R performing routing while the applications act on other packet headers.

9 Conclusion

We present D2R, a failover mechanism that leverages programmable switching technologies to perform routing completely in the data plane using P4. D2R is able to provide always-availability and policy-compliance under failures. We show how D2R can perform graph traversals while suffering a small number of recirculations. However, this is not instrinsic to our mechanism, rather dependent on the number of stages. If the number of stages doubled, recirculations would get halved. New programmable switches are getting more powerful, for instance, the new Barefoot Tofino 2 supports 12.8 Tbps line rate and has more stages for packet processing. Our work opens up a vast avenue of interesting open problems: Can we increase the coverage of policies we can implement in the data plane? Can we design hardware optimized for graph traversal to perform routing efficiently?

[6]We do not report actual numbers due to a confidentiality agreement with Barefoot.

References

[1] [n.d.]. BEHAVIORAL MODEL REPOSITORY. https://github.com/p4lang/behavioral-model.

[2] [n.d.]. BGP PIC Edge for IP and MPLS-VPN. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute$_{b}$gp/configuration/xe-16/irg-xe-16-book/bgp-pic-edge-for-ip-and-mpls-vpn.html.

[3] [n.d.]. IPv4 Loop-Free Alternate Fast Reroute. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute$_{p}$i/configuration/xe-3s/iri-xe-3s-book/iri-ip-lfa-frr.html.

[4] [n.d.]. Mininet. http://mininet.org/.

[5] [n.d.]. P4Studio. https://barefootnetworks.com/products/brief-p4-studio/.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM (SIGCOMM '16)*.

[8] 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* (Chicago, Illinois, USA) *(HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 121–126. https://doi.org/10.1145/2620728.2620746

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[10] 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* (Hong Kong, China) *(SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[11] Marco Chiesa, Andrei Gurtov, Aleksander Madry, Slobodan Mitrovic, Ilya Nikolaevskiy, Michael Shapira, and Scott Shenker. 2016. On the resiliency of randomized routing against multiple edge failures. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[12] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiundefinedski, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute: Hope for the Best and Program for the Worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3359989.3365410

[13] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. 2018. Andromeda: performance, isolation, and velocity at scale in cloud network virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 373–387.

[14] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide Configuration Synthesis. In *29th International Conference on Computer Aided Verification, Heidelberg, Germany, 2017 (CAV'17)*.

[15] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2013. FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 19–24.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, Berkeley, CA, USA, 51–64. http://dl.acm.org/citation.cfm?id=3307441.3307446

[17] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. 2005. Achieving sub-second IGP convergence in large IP networks. *ACM SIGCOMM Computer Communication Review* 35, 3 (2005), 35–44.

[18] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 163–174.

[19] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[20] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Measuring Control Plane Latency in SDN-enabled Switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. ACM, New York, NY, USA, Article 25, 6 pages. https://doi.org/10.1145/2774993.2775069

[21] 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)*. USENIX Association, Boston, MA, 161–176. https://www.usenix.org/conference/nsdi19/presentation/holterbach

[22] 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* (Hong Kong, China) *(SIGCOMM '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2486001.2486012

[23] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu

[24] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 3–14.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[27] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 103–115. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose

[28] D. Katz and D. Ward. [n.d.]. *Bidirectional Forwarding Detection (BFD)*. RFC 5880. https://tools.ietf.org/html/rfc5880

[29] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on* 29, 9 (october 2011), 1765 –1775. https://doi.org/10.1109/JSAC.2011.111002

[30] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. 2000. Delayed Internet routing convergence. *ACM SIGCOMM Computer Communication Review* 30, 4 (2000), 175–187.

[31] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving Convergence-free Routing Using Failure-carrying Packets. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Kyoto, Japan) *(SIGCOMM '07)*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/1282380.1282408

[32] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say {NO} to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 467–483.

[33] Junda Liu, Baohua Yan, Scott Shenker, and Michael Schapira. 2011. Data-driven network connectivity. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 8.

[34] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 399–412. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu$_{v}$incent

[35] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient Synthesis of Network Updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 196–207. https://doi.org/10.1145/2737924.2737980

[36] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[37] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2018. Hardware-Accelerated Network Control Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/3286062.3286080

[38] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. 2017. Decentralized Consistent Updates in SDN. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. ACM, New York, NY, USA, 21–33. https://doi.org/10.1145/3050220.3050224

[39] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 121–136.

[40] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2486001.2486022

[41] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Helsinki, Finland) *(SIGCOMM '12)*. ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/2342356.2342427

[42] G. Rétvári, J. Tapolcai, G. Enyedi, and A. Császár. 2011. IP fast ReRoute: Loop Free Alternates revisited. In *2011 Proceedings IEEE INFOCOM*. 2948–2956. https://doi.org/10.1109/INFCOM.2011.5935135

[43] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the Sixteenth ACM Workshop on Hot Topics in Networks*.

[44] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting Emerging Applications With Low-Latency Failover in P4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies* (Budapest, Hungary) *(NEAT '18)*. ACM, New York, NY, USA, 52–57. https://doi.org/10.1145/3229574.3229580

[45] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 164–176.

[46] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. ACM, New York, NY, USA, 213–226. https://doi.org/10.1145/2674005.2674989

[47] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables for Multi-tenant Networks. In *POPL*. ACM.

[48] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2018. Synthesis of Fault-Tolerant Distributed Router Configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 22.

[49] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. 2015. Central Control Over Distributed Routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/2785956.2787497

[50] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 5.

[51] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. 2017. Where has my time gone?. In *International Conference on Passive and Active Network Measurement*. Springer, 201–214.