ARDAPL: An Adaptive Resilient Data-Plane for Stateful Packet Processing

Chenxing Ji, Adrian Zapletal, and Fernando Kuipers
Delft University of Technology
{c.ji, a.zapletal, f.a.kuipers}@tudelft.nl

***Abstract*—Stateful network functions on programmable switches make decisions within microseconds by directly operating on in-switch state. However, their performance hinges on the availability and correctness of the state stored in scarce on-chip memory. When a switch failure occurs, fast reroute can restore connectivity, but the rerouted traffic loses access to the existing state, causing incorrect network behavior. In extreme cases, packets may no longer be forwarded. Given switch constraints and bandwidth overhead, maintaining synchronous backups of all states across all nodes is practically infeasible. Moreover, existing recovery mechanisms typically incur high latency. In this paper, we present ARDAPL, a framework that leverages unallocated hardware resources to achieve in-data-plane failover at the μs-scale. We have prototyped ARDAPL on Tofino switches and evaluated its impact on existing traffic. Our experimental results show that less than 0.002% of traffic was affected over the entire failure period, and ARDAPL can achieve sub-50μs failover time while incurring 5μs latency.**

***Index Terms*—Programmable Networks; Fault Tolerance; P4.**

I. Introduction

Programmable networking ASICs [1]–[4] have transformed the data-plane from a fixed-function forwarding pipeline into a target for deploying sophisticated network functions that operate at line-rate [5]. These network functions increasingly rely on in-switch states, such as registers and counters, to perform packet-level decisions. This stateful data-plane paradigm [6] enables capabilities that are infeasible with the previously used controller-centric Software-Defined Networking (SDN) alone. Applications of a stateful data-plane range from network telemetry and measurement [7], [8] to in-network computing and acceleration [9], as well as line-rate security applications such as [10], [11].

While they are useful and increasingly common, such stateful network functions introduce a pitfall that is easy to overlook: the correctness and service of a stateful data-plane program depend on the availability and correctness of the specific state-object it maintains locally. Device failures, reconfiguration, and link failures can render traffic unable to access the correct states [12].

For many applications [13]–[16], such a loss of state can be critical, potentially leading to the inability to forward flows and, thus, the disconnection of live traffic. Moreover, the outcome not only degrades the performance of the network but also causes the violation of the intended semantics, e.g., treating established flows as new flows, losing established policies. While fast failover mechanisms can redirect traffic in a timely manner [17], [18], fast failover per se does not preserve stateful information.

ISBN 978-3-903176-82-9

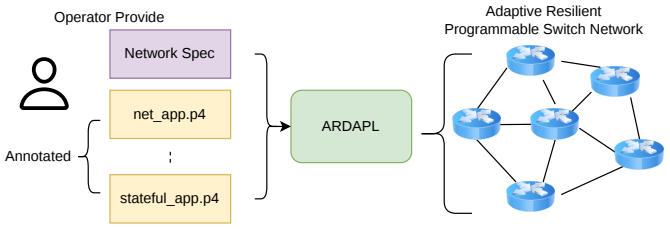

Fig. 1: High-level Overview of ARDAPL, where it takes in a network specification and a list of P4 programs with stateful applications, annotates these programs, and deploys them.

Prior work addresses important aspects of this problem, but falls short of providing rapid in-switch-state continuity under switch failures. Existing work on distributed abstractions and coordination frameworks [19], [20] simplifies the sharing and access of states across devices. Other works investigate the consistency of preserved states and their migration during planned updates, such as pipeline reconfiguration [21]–[23]. Fault tolerance has been explored through external recovery, in which the lost state can be reconstructed from outside the data-plane [12]. In contrast, fast reroute inside the data-plane operates on failure timescales that leave little room for controller-based or storage-based coordination. Reconstruction of the state or rolling back [24] can be expensive and may lag far behind data-plane failover, breaking the semantics of stateful network functions.

The scarce hardware resources, already heavily provisioned for forwarding logic, pose another challenge to achieving in-data-plane resilience. Even without failures, supporting state-intensive network functions is challenging due to limited hardware resources [25]. Therefore, simply replicating everything is infeasible in existing hardware. Recent work has investigated improving utilization through virtualizing on-chip state through logical address translation [26]. However, it does not provide a runtime resilience in the case of a switch failure. Instead, we approach this problem via slicing large state components into multiple contiguous entries.

In this paper, we present Adaptive Resilient Data-Plane (ARDAPL), a framework that provides μs-level failover time

through pre-allocated state backups. Figure 1 presents the high-level overview of ARDAPL. Our contributions are as follows:

1) We present ARDAPL, a three-switch, in-data-plane failover design that preserves index-equivalent state access after reroute via page/slice translation.
2) We introduce slice-based replica allocation that respects the per-stage resource and register-action constraints of switch pipelines.
3) We propose an adaptive policy that prioritizes which state to protect based on online read/write characteristics, enabling graceful resilience under tight SRAM/bandwidth budgets.
4) We prototype ARDAPL on Intel Tofino switches and evaluate the impact of failover on live traffic.

II. Background

Stateful network functions on programmable switches depend on in-switch state that is both scarce and pipeline-coupled [27]. When the state is lost or becomes inaccessible due to failures, the correctness of in-path forwarding decisions can be affected. Strategies that rely on out-of-data-plane devices for state recovery are often on a second scale [12]. In this section, we motivate the in-data-plane replication approach of ARDAPL and outline the associated challenges.

A. State Losses

A stateful data-plane application implicitly assumes that state is persistent and remains accessible throughout operations. In practice, states can be lost or invalidated due to (1) switch failures induced by power or hardware failures or (2) pipeline reloads. When the correct states are unavailable, the underlying network function may behave unpredictably. For example, an in-switch NAT application loses connection-tracking state, a load-balancer [13] yields under-performing balance, or a measurement network function reports incorrect estimations.

External state recovery can be used to restore utility, yet it can still be too slow at a timescale relative to transport and application performance. Redplane [12] showed that replicating the state to an external state-store can help preserve network correctness across failures, yet its end-to-end throughput still exhibits disruption on the order of a second during failure and recovery. Under the rapid development of high-performance networks, even second-level disruptions are large compared to typical RTT values, e.g., tens of milliseconds in WAN and sub-milliseconds in datacenters [28]. This failover time window can trigger TCP retransmission timeouts and backup mechanisms, reducing the congestion window and degrading loss-based TCP throughput beyond the failure event interval [29].

More importantly, within this time window, the network may violate policy, degrade service quality, and similar issues. Recent work also highlights that the impact of state loss is not uniform across components and applications [12], [30], motivating selective protection and rapid response.

Examples	Reconstructability	Priority	R/W Ratio
Counters, Sketches	High	Low	Low
Rates, Thresholds	Medium	Medium	Medium
NAT, Firewall	No	High	High

TABLE I: Different data-plane state types. Reconstructability is the ability to reconstruct the state within the data plane from observing ongoing traffic, and replication priority denotes how the priority should be assigned to each type.

1) Types of Stateful Network Functions: The resilience requirements of a stateful network function are impacted by two major factors: (1) how its states influence forwarding decisions and (2) whether the state can be reconstructed from live traffic. We categorize the existing data structures in Table I. State components with a higher Read/Write ratio tend to be more critical for flow consistency.

B. Slicing

Replication granularity needs to match the hardware constraints. Per-entry replication is impractical since it requires fine-grained allocation and a large mapping to allocate each cell. Conversely, replicating the entire stateful component is often wasteful because stateful access is typically locality-sensitive: a small subset of the index remains active; even a hash-indexed structure exhibits hot buckets under skewed traffic. These observations motivate a middle ground: replicating at the slice granularity to achieve practical protection within tight memory and bandwidth budgets.

C. Why in-data-plane replication is hard

Achieving resilience using in-data-plane replicas introduces the following challenges.

State-access semantics after rerouting. Upon failure, packets are redirected to a different switch. The correctness of the replica then depends on preserving the state-access semantics, and the rerouted packet must be able to access the entry corresponding to the primary's index. This requirement is stronger than merely storing a copy of the state, because an incorrect mapping can disrupt network behavior.

Replication under pipeline budget. Programmable switches [1] offer limited memory ($\sim$ hundreds of Mb) and a bounded number of stateful operations per packet. Existing work demonstrates the feasibility of data-plane state sharing and replication [19], [26], [31], but highlights the trade-off between overhead, placement, and guarantees.

Replica placement cost. The placement needs to jointly optimize reroute latency and the communication overhead brought by synchronization. Placement and communication costs strongly shape the consistency bounds and failover latency a system can provide [32], [33]. This motivates a need for topology-aware replica placement.

Consistency under failures. Stateful replicas are only useful if they are sufficiently consistent with the primary to support the correct failover. As discussed in Section II-A1, bounded staleness suffices in many cases [34]. Therefore, a mechanism

for selecting the consistency level of a stateful component is necessary.

III. Failure Model

In this paper, we consider a network of programmable switches capable of executing stateful network functions in the data-plane. A stateful component is any data structure whose contents can influence packet-processing behavior, e.g., registers or match-action state. For the selected component, ARDAPL pre-allocates replica slices on other switches using available hardware resources and transforms the data-plane program to support correct replica-index access.

A. Failure Types

We assume a single-switch fail-stop model in which at most one switch fails at a time, becoming unreachable or unable to process packets due to hardware failures, power loss, or similar causes. Such failure renders both the network function logic and its locally stored state unavailable. This captures the common case of localized device outages observed in production networks [35]. Link failures can be handled by mechanisms such as [17]. We do not consider multiple failures and correlated failures.

B. Correctness Criterion

Our objective is to preserve the correct network function behavior under the failure described above. We express this as per-flow state safety: for any packet that accesses a stateful component at logical index i of the primary switch, it should access the corresponding entry for the same flow after rerouting. We allow bounded deviation between the primary and replica states depending on the application. Specifically, measurement-state tolerates bounded staleness, whereas semantics-critical per-flow state requires staleness that preserves policy correctness.

IV. Design

ARDAPL preserves stateful forwarding semantics across a primary switch with fail-stop failures by pre-allocating selective state replicas in unallocated stateful resources and enforcing index access equivalence. Figure 2 summarizes the packet flows and the division of the responsibilities among the three roles:

1) **The Primary Switch:** executes the original network function and emits synchronization packets for write operations to the protected slices.
2) **The Upstream Switch:** detects a failure and redirects affected traffic. In addition, the upstream switch recomputes the logical state index that the primary switch would have used and emits it to the replica switch.
3) **The Replica Switch:** hosts virtualized memory pools for replicated slices and translates the carried logical index into a physical location. It also executes the same state-dependent logic to ensure forwarding correctness.

ARDAPL operates in two phases, provisioning and runtime operation. During provisioning, it uses the given topology and

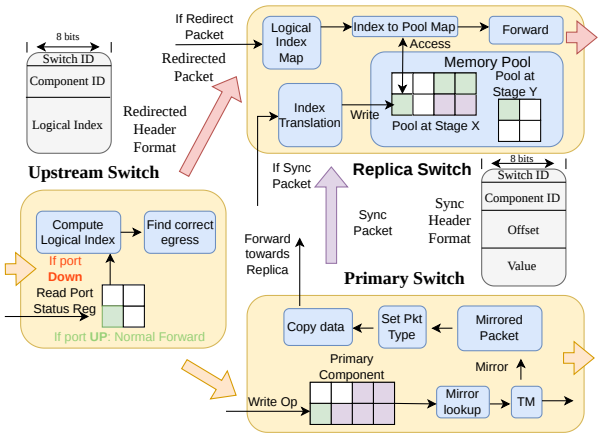

Fig. 2: Data-plane design of synchronization between the primary component and the replica slice in the memory pool, as well as how traffic flows with and without failures.

annotations to (1) determine which stateful components require replication and which operation mode they need, and (2) initialize replica pools using the available per-stage resources described in Section IV-B2. During runtime operation, if a failure happens, packets are redirected to the correct replica switch without the direct involvement of the controller in the critical path.

A. Data-plane Logic

We now describe the per-packet processing logic on the normal path and on the failover path, as illustrated in Figure 2. In a normal processing scenario, packets are forwarded and processed by the primary switch. Upon failure detection at the upstream switch, packets are redirected. The remainder of this subsection will explain the data-plane processing logic of each switch in detail.

1) Upstream switch: The upstream switch enables μs-scale in-data-plane fast failover by redirecting traffic whose next hop is a failed primary. To achieve this, the upstream maintains a dedicated register for monitoring the port status of each connected switch. Such a register can be maintained by any underlying failure detection mechanism, and ARDAPL only consumes the read value. Thus, ARDAPL is independent of the underlying failure detection mechanism.

Upon redirecting packets under failure, the upstream switch recomputes the same logical index that the primary switch would use for the replicated component. The steering table maps $s_p, C_{id}, \mathsf{page_{id}}$ intervals to the correct egress port of the replica switch that hosts the corresponding slice. Finally, it attaches s_p, C_{id}, I in the redirect header so that the replica can access the correct state entry after rerouting.

2) Primary switch: The primary switch runs the original network function logic and maintains the authoritative copy of the stateful component. In addition, ARDAPL modifies the original P4 program to expose state modifications as explicit events and generate synchronization updates towards the allocated replica pool, as illustrated in Algorithm 1.

Algorithm 1 Data-plane replication logic of a primary switch, where green lines are replication code inserted by ARDAPL and black lines represent the original state write operation.

1: **procedure** INGRESS($p \leftarrow hdr$,$meta$, s_p,C_{id})
2: meta.rv = C_{write}.execute(I) ▷ Write Action
3: $mirror\_lookup(s_p, C_{id}$,I)
4: copy_into_bridge_header(s_p, C_{id},I, meta.rv)
5: **end procedure**
6: **procedure** EGRESS($p \leftarrow hdr$,$meta$)
7: **if** meta.mirrored **then**
8: copy_bridge_header_into_header();
9: hdr.ethernet.etherType = ETHTYPE_SYNC;
10: **end procedure**

Concretely, when a packet performs a write operation C_{write} to a protected component entry, the primary switch records the component identifier and the index. Then it emits a synchronization packet to the corresponding replica via a mirroring session. The synchronization header carries the primary switch id s_p, component id C_{id}, the entry offset I, and the new value, depicted in Figure 2. When a single packet updates multiple entries, for example, sketch-based components [8] typically apply multiple hashes to reduce collisions, the primary switch aggregates multiple synchronization headers by including a small counter and an array of synchronization headers. The synchronization emission rate is managed by the component's synchronization policy and the replication bandwidth budget described in Section IV-D and Section IV-C.

3) Replica Switch: A replica switch hosts a virtualized memory pool built from unallocated per-stage resources. The memory pool stores abstracted replica slices and runs the corresponding NF logic to produce the correct forwarding decision for redirected traffic.

Upon receiving a packet, the replica switch first computes the physical pool location based on the logical index value in the packet header, as illustrated in Lines 10 - 14 of Algorithm 2. It relies on match-action entries populated when a slice is allocated. First, a translation step derives the logical index I into the $page_{id}$. Then, the pool map lookup keyed by $(s_p, C_{id}, page_{id})$ returns $base$ and the $pool_{id}$, which $pool_{id}$ maps to a pool located at a specific stage, and $base$ is the starting physical index of that slice. The actual physical index of that state is then computed using the lowest 12 bits of the logical index I and the base.

For state entry access, if the incoming packet is a synchronization packet from the primary switch, the replica switch simply writes to the corresponding entry in the register pool. For the redirected packets, the replica switch executes the same state-dependent forwarding logic as the primary to ensure forwarding consistency.

B. Virtual Memory Pool and Slicing

ARDAPL employs a memory abstraction to enable practical in-data-plane failover under pipeline constraints, where state-

Algorithm 2 Data-plane replication synchronization and redirect traffic index-mapping algorithm on a replica switch.

Input: $p \leftarrow$ **hdr**
1: **procedure** INGRESS
2: **if** hdr.sync.isValid() **then**
3: index_translate() ▷ Compute Physical index
4: pool.write($index$) ▷ Write value to the index
5: **else if** hdr.redirected.isValid() **then**
6: index_translate() ▷ Compute Physical index
7: access_state()
8: forwarding() ▷ Same logic as primary
9: **end procedure**
10: **procedure** INDEX_TRANSLATE(s_p,C_{id},I)
11: $page_{id} \leftarrow$ SHIFTRIGHT(I, 12)
12: Δ_{page}, $pool_{id}$ = pool_map(s_p,C_{id},$page_{id}$)
13: $page \leftarrow \Delta_{page} + page_{id}$
14: $index \leftarrow$ SHIFTLEFT($page$, 12) + (I & $0xfff$)
15: **end procedure**

ful resources are fragmented across stages and support only limited operations. ARDAPL achieves this by aggregating the available per-stage resources into replica pools that provide contiguous logical address space and expose supported actions.

1) Slicing and Page-based Addressing : For a logical component C_{id} on a primary switch s_p with size of y, ARDAPL partitions the memory space of $[0, y]$ into slices and has the ability to allocate each slice as a contiguous region inside a replica pool on a replica switch s_r. We can calculate a contiguous number of pages with:

$$numPages = \left\lceil \frac{y}{\text{page\_size}} \right\rceil \tag{1}$$

Each slice allocation is described by the following:

$$slice \leftarrow \langle s_p, C_{id}, [P_s, P_e], s_r, pool_{id}, base \rangle \tag{2}$$

, where $pool_{id}$ can identify the pool allocated on a specific stage, and $base$ is the physical starting index of the slice.

To enable the slicing mechanism on programmable hardware, ARDAPL uses a two-level address space. The upstream switch splits a logical index I into an upper $page_{id}$ and a lower $page\_offset$. Therefore, the upstream switch can steer the redirected packets using a range-match table based on the I to the corresponding egress port of the replica. The replica switch then uses $pool\_map$ to resolve $(s_p, C_{id}, page_{id})$ into $base$ and $pool_{id}$.

2) Pool Capability Provisioning: A replica switch exposes a virtual register pool backed by one or more physical register components. In the pipeline design, each stateful component has limited capabilities. For example, each physical register array can be paired with a small number of stateful operations [1], which constrains the set of operations that can be executed for a given pool. ARDAPL defines this set of action capabilities at compile time, given the network specification.

For each stateful component C, its potential replica slice must support two classes of operations:

1) Runtime operations that will be executed by the redirected packets when they access C.
2) Synchronization operations that are required to write the synchronization value from a primary to its replica.

Therefore, the effective requirement of a primary slice to be able to be allocated on a replica slice is:

$$Ops(C) = Ops_{\text{nf}}(C) \cup Ops_{\text{sync}}(C),$$

where $Ops_{\text{nf}}(C)$ captures the access primitives of the component, e.g., read, write, etc., and $Ops_{\text{sync}}(C)$ is the operation to write the updated value. In addition, ARDAPL also records the maximum number of stateful actions executed per packet over a failover path, which must remain under target limits.

ARDAPL associates each pool P with a capability set $Ops(P)$ determined by the set of register actions instantiated for the physically allocated register array. Rather than instantiating all possible actions, ARDAPL provisions a small set that the slice allocation and placement algorithm described in IV-B3 uses for the allocation of a replica slice.

Pool capability provisioning must also respect network-level failover performance, such that the redirected traffic incurs minimal detour cost. ARDAPL limits the candidate replicas by only provisioning the failover replicas to a small set of switches on fast failover paths, which is typically within a few hops with similar latency. Given this set S, ARDAPL tries to provision pool capabilities such that for every potentially protected component, at least one low-latency candidate replica switch exposes a pool where $Ops(C) \subseteq Ops(P)$. Such an approach decouples the compilation-time constraints from the runtime allocation algorithm, while ensuring that rerouted packets can be served without incurring the overhead of stretched rerouting paths.

3) Slice Allocation and Placement: ARDAPL performs replica allocation and placement at slice granularity. Allocation determines which logical page ranges of a primary component are protected, and placement determines where each protected slice resides by binding it to a region within a memory pool on a replica switch. It takes the following input: (1) a list of slice requests $\mathcal{S}$ where a slice is defined by Equation 2, (2) candidate replica switches $\mathcal{R}(s_p)$ selected for a given primary switch s_p based on topology and pool feasibility constraints, and (3) the set of pool maps available on each replica, where each pool $\mathcal{P}$ contains the information of unallocated regions and its capabilities $Op(\mathcal{P})$.

The allocation placement is described in Algorithm 3. For each slice, it scans candidate replicas and their pools, filters pools that can support the slice (Line 5), and uses a first-fit algorithm to reserve contiguous pages. Upon success, it installs an upstream switch control-plane rule that redirects packets for the slice to the selected replica switch, and a replica switch mapping rule that maps the logical page id to the chosen pool and base offset within that pool.

Algorithm 3 Slice allocation and placement.

Require: $\mathcal{S}$, $\mathcal{R}(s_p)$, $\mathcal{P}(s_r) \leftarrow \langle pool_{id}, Op(\mathcal{P}), List_{free}(P)\rangle$
1: **for each** $s \in \mathcal{S}$ **do**
2: $numPages \leftarrow P_e - P_s$; $placed \leftarrow false$
3: **for each** $s_r \in \mathcal{R}(s_p)$ **do**
4: **for each** pool $P \in \mathcal{P}(s_r)$ **do**
5: **if** $Ops(s) \subseteq Ops(P)$ **then**
6: $base \leftarrow$ FIRSTFIT($freeList(P)$, $numPages$)
7: **if** $base \neq \perp$ **then**
8: ALLOC(freeList(P), [base, base+$numPages$])
9: // Compute constant for page-based addressing
10: $\Delta_{page} \leftarrow base - P_s$
11: // Install rules on the upstream switch
12: $(s_p, comp_{id}, page_{id}) \rightarrow replica\_eg\_port(s_r)$
13: // Install rules on the replica switch
14: $(s_p, comp_{id}, page_{id}) \rightarrow \langle pool_{id}(P), \Delta_{page}\rangle$
15: $placed \leftarrow true$;
16: **break**
17: **if** $placed$ **then break** ▷ go to next slice allocation

C. Synchronization Policies

Data-plane stateful components with a high read/write ratio may perform write operations for each packet processed, causing large synchronization overhead. To reduce the bandwidth overhead of in-data-plane replication, ARDAPL is inspired by [34], and supports two synchronization policies for replicated slices: `Exact` and `Bounded`. The policy of each component and its parameters are determined by the provided annotations as well as the resource-aware controller described in Section IV-D. Both policies use the same header format to describe a synchronization operation.

1) Exact Synchronization: For `Exact` components, the primary emits an update record for every state write that targets a protected slice, and the replica applies the record after index translation. This provides the strongest continuity across failover, but the traffic overhead scales with the component write rate; therefore, ARDAPL reserves `Exact` for operator-marked decision-critical state (§IV-D).

2) Bounded Staleness Synchronization: For `Bounded` components, ARDAPL reduces synchronization traffic, while bounding replica divergence using a divergence and threshold design in the spirit of ApproSync [34]. Instead of maintaining per-index `last-sent` values for the entire component, the primary keeps a fixed-size on-chip hash table keyed by $(C_{id}, logical\_index)$. Each entry stores an index tag for validity checking and a value of `last_sent`. On each write to the component C producing a new value v, the primary probes the bucket and emits a synchronization packet if $|v - \texttt{last\_sent}| \geq t_C$, then writes the value to the `last_sent`. When an index value mismatch occurs, the entry is replaced. The controller tunes t_C to respect the per-component budget B_C using the monitoring mechanism described in Section IV-D.

D. Adaptive Replication Decision

Due to switch resource and bandwidth constraints, replicating all state everywhere is practically infeasible. Therefore, ARDAPL makes replication a selective decision on

which stateful components (or slices) are worth protecting and chooses an appropriate replication mode under a given budget. ARDAPL leverages lightweight operator annotations per stateful component to guide these replication decisions.

Adaptive replication is designed for bounded synchronized objects. Unlike critical states that require `Exact` replication, `Bounded` objects can tolerate limited divergence, which allows ARDAPL to trade synchronization and memory overhead for resilience.

1) Dynamic Slice Access Monitoring: To generate slice replication requests under resource constraints, ARDAPL monitors reads and writes of stateful components at slice granularity. On the primary switch, ARDAPL inserts a table that is dedicated to count state access via `DirectCounters`. Each slice needs two entries in this table to track `read` and `write` operations separately. Both `DirectCounters` will be used to calculate the state's importance. More importantly, the control-plane can add and/or delete table entries at runtime to monitor only a subset of slices, e.g., those currently allocated to a replica switch. Such an approach keeps the dynamic slice-access monitoring footprint proportional to the number of monitored slices rather than the total number of component sizes on a given switch, which is crucial for resource-scarce programmable hardware.

2) Resource-aware Slice Replication: For bounded components, ARDAPL selects the replicated slice using access rates measured over an interval of time i.

$$\lambda_r(C,s) \triangleq \frac{N_r(C,s)}{i}, \qquad \lambda_w(C,s) \triangleq \frac{N_w(C,s)}{i},$$

, where $N_r(C,s)$ and $N_w(C,s)$ are the number of read and write operations observed for that slice in the given time slot. $\lambda_r(C,s)$ captures how hot this slice is during this period of current traffic, while $\lambda_w(C,s)$ depicts the amount of synchronization overhead that replicating this slice will cost.

Therefore, we use the following utility function to rank each replicated slice, and the constant 26 represents the packet size of a synchronization packet (Ethernet header size + synchronization header):

$$score(C,s) = \frac{\lambda_r(C,s)}{\epsilon + 26 \cdot \lambda_w(C,s)} \tag{3}$$

The intuition behind such a utility function is that this equation yields the slice that maximizes benefit per unit of replication bandwidth cost.

3) Bandwidth Budgets: We express the budget for bounded components as an allowed emission rate for synchronization. Let C_{sync} be the bandwidth reserved for synchronization traffic on the path from a primary to its replica. For each component C, let S_C denote the size of the synchronization packet. The control-plane assigns each bounded component a priority weight w_C that is parsed from the operators' annotation, then uses such w_C to divide the available bandwidth proportionally:

$$B_C = \frac{w_C}{\sum_{C' \in \mathcal{C}_{Bounded}} w_{C'}} \cdot \frac{C_{\text{sync}}}{s_C}, \tag{4}$$

Therefore, the B_C represents the maximum emission rate per second of a component C for bounded synchronization.

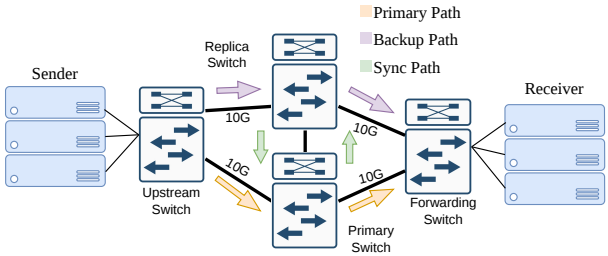

Fig. 3: Evaluation topology.

E. Program Transformation

ARDAPL implements the in-data-plane failover and the adaptive replica pool access by automatically transforming the provided P4 program.

Given the annotated P4 programs and a network specification, ARDAPL produces (1) an instrumented primary program that emits replica updates, (2) a replica program that allocates the unallocated hardware resources as a memory pool for dynamic replica allocations, and (3) an upstream steering data-plane program that also computes the logical indices for failover packets. The program transformation process of ARDAPL also generates the corresponding control-plane table rules that populate the slice-steering mechanism and index translation tables when slices are allocated.

For a given component C in the primary data-plane program, the program translation extracts two parts and places them on an upstream switch and a potential replica switch, respectively, as explained in Section IV-A:

1) The logic for computing the logical index.
2) The logic for accessing and using the state stored.

To guarantee index-equivalence and correct steering of the packets, ARDAPL must ensure that packets access the same logical state entry across the primary and replica switches. Many data-plane components derive their logical index from packet headers via hashing, slicing, or masking. ARDAPL extracts such logic into a reusable code block, ensuring the index equivalence across switches. The upstream uses this block to recompute the same logical index and attach to the emitted headers when redirecting packets.

On the primary, ARDAPL inserts logic (green in Algorithm 1) that converts write operations into synchronization headers and send to the replica switch via a mirroring session. On the replica, ARDAPL injects logic for an indirect memory pool access, which looks up a slice match-action table and translates the logical index into (1) a pool number to identify the pool stage, (2) a physical offset of the pool's index.

V. Evaluation

We implemented our ARDAPL design and evaluated its failover time using 4 BF2556x-1T hardware Tofino switches [36] with the topology depicted in Figure 3 over 10Gbps links. Sender and receiver of the traffic are servers running Ubuntu 20.04 LTS with Intel Xeon CPUs.

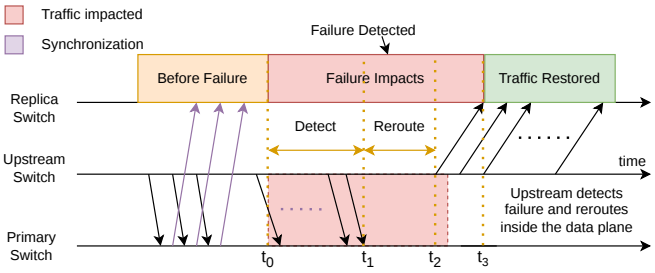

Fig. 4: Failover timeline and decomposition.

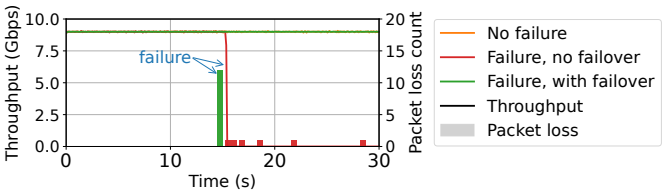

Fig. 5: The impact of failure on the existing TCP connection over a stateful firewall w and w/o the failover of ARDAPL.

A. Failure and Recovery

ARDAPL currently protects a failed primary switch under the single-switch failure model described in Section III. To demonstrate the fast failover capability of ARDAPL, we conducted experiments to measure the impact on ongoing traffic. During the experiment, we used the failure detection mechanism developed in [30], achieving sub-10μs failure detection, ensuring a low failure detection time, and enabling ARDAPL to perform failover in a timely manner. To demonstrate the effectiveness of state synchronization, we implemented an in-data-plane stateful firewall application that establishes the connection state upon receipt of a TCP SYN packet on the primary switch. Therefore, the recognized traffic can be forwarded, and unrecognized traffic is blocked. To demonstrate that ARDAPL can perform failover while maintaining line rate, we initiate traffic at link speed from the sender to the receiver. The traffic was generated using iperf3 with CUBIC as the TCP congestion control.

Fig. 5 depicts the throughput and the packet loss count of the line-rate traffic over 30 seconds in three scenarios. Without failure, traffic follows the primary path, and ARDAPL redirects it to the backup path via the replica switch. The no-failure scenario serves as a baseline. In the two failure scenarios, failure occurs at approximately 15s with the ARDAPL failover mechanism and at 15.5s without our failover mechanism.

Without the ARDAPL failover mechanism, there is no equivalent state that can forward traffic, resulting in a 100% loss of traffic volume upon failure. With ARDAPL enabled, 12 packets are retransmitted, indicating that 12 packets are affected by the failover, corresponding to less than 0.002% traffic affected. This almost negligible failover effect demonstrates the effectiveness of ARDAPL, which achieves fast failover entirely within the data-plane.

Figure 4 represents the failover timeline and decomposition.

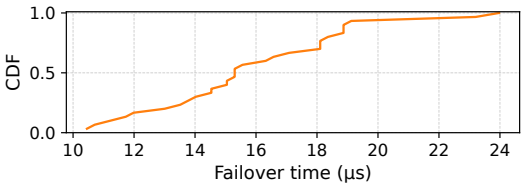

Fig. 6: Failover time over 30 distinct experiments, where failure happens during a line rate TCP flow is being transmitted.

t_0 represents the time when the failure happened, t_1 is the time when the failure is detected by the detection mechanism, and t_2 is the first packet that is rerouted via the backup path, t_3 is when the first packet arrives in the data-plane of the backup switch. Therefore, the failover time can be calculated using Equation 5, where t_{detect} can be expressed by $t_1 - t_0$, $t_{reroute}$ equals $t_2 - t_1$, and t_{prop} is equivalent to $t_3 - t_2$.

$$T_{failover} = t_{detect} + t_{reroute} + t_{prop} \tag{5}$$

We measure failover time at the forwarding switch using its data-plane timestamps as the interruption gap between the first packet arriving on the backup path and the last packet arriving on the primary path immediately before that event. Figure 6 depicts the failover time measured inside the forwarding switch over 30 different experiments, and shows that ARDAPL can achieve as low as 10μs failover time, as well as maintaining the failover time to be below 30μs. The difference in the measured failover time is due to differences in the failure detection mechanism and the inter-transmission gap of TCP flows.

B. Resource Consumption and Overhead

Since hardware programmable switches typically have a constrained number of stateful operations associated with each stateful component per stateful register [1], it is important to demonstrate that ARDAPL's replica pool design supports the same processing logic as the original component, i.e., most register access logic needs less than 3 stateful actions.

We examined a list of open-source Tofino-based projects[1] from multiple publicly available implementations. Figure 7 demonstrates the number of actions per register components. The results show that the majority of the components ($\sim 80\%$) contain 2 actions, and around 20% of the components in existing projects contain 1 stateful action block. This demonstrates that there are enough action slots for the virtualized memory pools to allocate multiple actions together.

In addition, we show that ARDAPL incurs little processing overhead by measuring the round-trip time of packets from the sender to the receiver. The experiments were conducted over 500 sample packets, and the results are depicted in Figure 8. The results indicate that, compared to the scenario where ARDAPL is not applied, ARDAPL incurs a 5μs latency overhead for packet processing in the median.

[1]The list contains 37 P4 Tofino programs and was primarily gathered from repositories listed at https://github.com/Princeton-Cabernet/p4-projects.

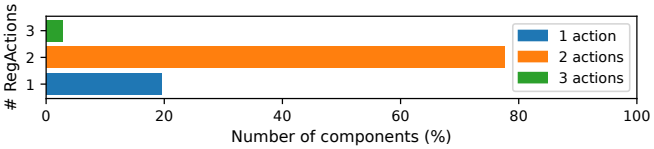

Fig. 7: Number of actions with stateful components in various open-source Tofino projects.

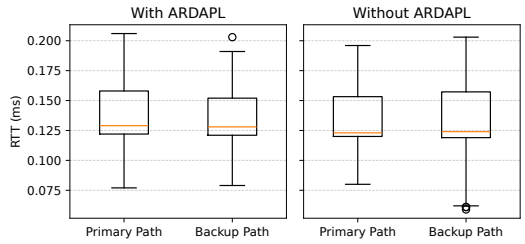

Fig. 8: RTT between sender and receiver over primary path and backup path in Figure 3 with and without ARDAPL.

VI. Related Work

A. Network Resilience and Fast Reroute

Prior work shows that substantial resilience mechanisms can be implemented inside the data-plane. Blink [37] demonstrates fast flow connectivity recovery by analyzing TCP behavior to directly react to failures in the data-plane, enabling fast rerouting within the switch pipeline. InFaRR [18] similarly investigates an in-network fast rerouting mechanism and emphasizes minimizing recovery latency within the network. Other systems, such as [17], specifically target low-latency failover behaviors in P4, while SQR [38] targets packet-loss recovery under link failures to improve network reliability. The aforementioned work primarily addresses connectivity and packet delivery under failures; however, ARDAPL emphasizes stateful forwarding logic after failover, where loss of in-data-plane state can directly affect network correctness.

B. Abstractions and Fault Tolerance

Domino [39] proposed a packet-transaction abstraction that improves stateful logic programmability; however, it does not account for state loss due to failover. SwiSh [19] proposes a distributed shared-state abstraction that highlights the high-latency problem of control-plane state access relative to in-data-plane state access and motivates data-plane-centric designs. Redplane [12] targets fault tolerance in the in-switch stateful application by providing a fault-tolerant state access and maintenance mechanism that continues to function under failures, leveraging external devices. ApproSync [34] focuses on approximate synchronization between the data-plane and the control-plane to reduce the overhead of exporting in-switch states for network management and monitoring purposes. ARDAPL differs from the aforementioned work and targets in-data-plane state continuity for latency-sensitive packet-processing logic by provisioning replica slices and selectively maintaining them within the data-plane.

C. Replica Optimization

Muqaddas et al. [31] formalized optimal state-replication placement using an ILP and discussed the trade-off between placement and overhead. However, real hardware targets often impose strict resource and compilation constraints [25], [26], [40]. Therefore, replication decisions for resilience must be co-designed with state distribution and the correct processing logic that consumes the state. Recent work [41] studies the co-design for deploying complex stateful network functions across hardware and software components. ARDAPL builds on the previous directions by coupling replication with data-plane failover and shaping consistency choices to the semantics and constraints of in-switch states. ARDAPL adopts first-fit slice allocation to keep allocation lightweight and efficient, while future work can investigate more advanced algorithms for optimal replica-slice placements.

D. Dynamic Memory Allocation and Virtualization

Memory virtualization is a key technique for flexible placement and mapping of in-switch state. NetVRM [26] introduces virtual register arrays that can be translated to physical arrays, enabling flexible allocation and increasing overall application memory utilization. FlxVRM [42] extends memory virtualization towards online configuration inside the data-plane. MP5 [43] proposed a multi-pipeline architecture for executing stateful applications across multiple pipelines while preserving correctness for shared states. Other systems treat in-network state and compute as a shared resource layer for specific workloads [44], [45]. ARDAPL is orthogonal to these works and leverages virtualization and dynamic allocation as mechanisms to provision replica slice allocations under strict pipeline constraints.

VII. Discussion

A. State Correctness

ARDAPL incorporates bandwidth consumption mechanism described in Section IV-C for components with high update rates. However, synchronization packets traveling inside the data-plane may be dropped, or reordered, and ARDAPL does not implement a mechanism for reliable data-plane state transfer. Instead, the existing mechanisms proposed in [12], [46] can be incorporated in ARDAPL in the future and ensure synchronization packet delivery for critical components.

B. Resource Usage

ARDAPL consumes additional PHV bits, pipeline stages, SRAM, and bandwidth to maintain replicas. Such overhead is manageable because the provided resilience is adaptive to the consumption on each component, and only the critical states use the strict replication policy. Under tight constraints and budgets, ARDAPL can be tuned to select a small component set to maintain the resilience of the stateful network function.

VIII. Conclusion

We present ARDAPL, an adaptive resilience framework that preserves state-access semantics across fail-stop failures by multi-switch failover, virtualized replica pools, and resource-aware synchronization. On Intel Tofino hardware with a stateful firewall under line-rate TCP traffic, ARDAPL demonstrates sub-50 μs failover, and less than 0.002% of traffic was affected during the failure period, while incurring 5μs latency.

References

[1] Intel, "Intel Tofino Intelligent Fabric Processors."

[2] Broadcom, "Broadcom trident 4."

[3] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, "Using trio: juniper networks' programmable chipset - for emerging in-network applications," in *SIGCOMM*, 2022.

[4] AMD, "Amd pensando dpu accelerators."

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *SIGCOMM Comput. Commun. Rev.*, aug 2013.

[6] X. Zhang, L. Cui, K. Wei, F. P. Tso, Y. Ji, and W. Jia, "A survey on stateful data plane in software defined networks," *Computer Networks*, 2021.

[7] S. L. Feibish, Z. Liu, N. Ivkin, X. Chen, V. Braverman, and J. Rexford, "Flow-Level Loss Detection with Δ-Sketches," in *SOSR*, 2022.

[8] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *SIGCOMM*, 2016.

[9] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of SOSP '17*, ACM, 2017.

[10] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *NDSS*, 2020.

[11] E. Bardhi, C. Ji, A. Imran, M. Shahbaz, R. Lazzeretti, M. Conti, and F. Kuipers, "O'mine: A novel collaborative ddos detection mechanism for programmable data-planes," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, pp. 771–788, IEEE, 2025.

[12] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan, "Redplane: Enabling fault-tolerant stateful in-switch applications," in *SIGCOMM*, 2021.

[13] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *SIGCOMM*, 2017.

[14] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4Knocking: Offloading host-based firewall functionalities to the network," in *ICIN*, 2020.

[15] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware," in *IFIP Networking*, 2020.

[16] W. Wu, Z. Li, X. Liu, Z. Wang, H. Pan, G. Zhang, and G. Xie, "Lemon: network-wide ddos detection with routing-oblivious per-flow measurement," in *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, (USA), USENIX Association, 2025.

[17] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting Emerging Applications With Low-Latency Failover in P4," in *Proceedings of NEAT*, 2018.

[18] F. L. Verdi and G. V. Luz, "Infarr: In-network fast rerouting," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2319–2330, 2023.

[19] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "SwiSh: Distributed shared state abstractions for programmable switches," in *NSDI*, Apr. 2022.

[20] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing," in *SIGCOMM*, 2016.

[21] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *SOSR*, 2017.

[22] J. Xing, A. Chen, and T. S. E. Ng, "Secure State Migration in the Data Plane," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020.

[23] C. Ji and F. Kuipers, "State4: State-preserving reconfiguration of p4-programmable switches," in *NetSoft*, 2023.

[24] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-Recovery for Middleboxes," in *SIGCOMM*, 2015.

[25] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *SIGCOMM*, ACM, 2020.

[26] H. Zhu, T. Wang, Y. Hong, D. R. K. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual Register Memory for Programmable Networks," in *NSDI 22*, Apr. 2022.

[27] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 193–207, 2022.

[28] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM Comput. Commun. Rev.*, Aug. 2010.

[29] W. Eddy, "Transmission Control Protocol (TCP)." RFC 9293, Aug. 2022.

[30] C. Ji and F. Kuipers, "Fastr: Fast resilience for stateful programmable data planes," in *2025 21st International Conference on Network and Service Management (CNSM)*, 2025.

[31] A. S. Muqaddas, G. Sviridov, P. Giaccone, and A. Bianco, "Optimal State Replication in Stateful Data Planes," *IEEE Journal on Selected Areas in Communications*, 2020.

[32] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 239–282, 2002.

[33] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, *et al.*, "drmt: Disaggregated programmable switching," in *SIGCOMM*, pp. 1–14, 2017.

[34] X. Chen, Q. Huang, D. Zhang, H. Zhou, and C. Wu, "ApproSync: Approximate State Synchronization for Programmable Networks," in *ICNP*, 2020.

[35] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM*, 2011.

[36] "Advanced Programmable Switches (APS), BF2556X-1T Advanced Programmable Switch, APS Networks."

[37] T. Holterbach, "Blink: Fast connectivity recovery entirely in the data plane," in *NSDI*, p. 84, 2019.

[38] T. Qu, R. Joshi, M. C. Chan, B. Leong, D. Guo, and Z. Liu, "Sqr: In-network packet loss recovery from link failures for highly reliable datacenter networks," in *ICNP*, 2019.

[39] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *SIGCOMM*, 2016.

[40] Z. Chen, Y. Feng, S. Liu, H. Song, H. Zhou, T. Yun, W. Xu, T. Pan, and B. Liu, "Optimusprime: Unleash dataplane programmability through a transformable architecture," in *SIGCOMM*, 2024.

[41] Y. Yuan, W. Wu, X. Yao, R. Chen, and G. Zhang, "Network functions with dynamic state management on programmable switches," *IEEE Transactions on Networking*, pp. 1–13, 2025.

[42] M. Qian, L. Cui, F. P. Tso, Y. Deng, Z. Zhang, and W. Jia, "Flxvrm: Enabling online configuring memory via virtualization on programmable data plane.," *IEEE Transactions on Services Computing*, 2025.

[43] V. Shrivastav, "Stateful multi-pipelined programmable switches," in *SIGCOMM*, 2022.

[44] H. Wang, D. Sun, J. Hu, and K. Chen, "Enabling in-network acceleration over the cloud," in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pp. 1–10, IEEE, 2025.

[45] B. Zhao, C. Liu, J. Dong, Z. Cao, W. Nie, and W. Wu, "Enabling switch memory management for distributed training with in-network aggregation," in *IEEE INFOCOM*, pp. 1–10, IEEE, 2023.

[46] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 35–49, USENIX Association, Apr. 2018.