Swing State: Consistent Updates for Stateful and Programmable Data Planes

Shouxi Luo* Hongfang Yu
University of Electronic Science and Technology of China

Laurent Vanbever
ETH Zürich

ABSTRACT
With the rise of stateful programmable data planes, a lot of the network functions that used to be implemented in the controller or at the end-hosts are now moving to the data plane to benefit from line-rate processing. Unfortunately, stateful data planes also mean more complex network updates as not only flows, but also the associated states, must now be migrated consistently to guarantee correct network behaviors. The main challenge is that data-plane states are maintained at line rate, according to possibly runtime criteria, rendering controller-driven migration impossible.

We present Swing State, a general state-management framework and runtime system supporting consistent state migration in stateful data planes. The key insight behind Swing State is to perform state migration entirely within the data plane by piggybacking state updates on live traffic. To minimize the overhead, Swing State only migrates the states that cannot be safely reconstructed at the destination switch.

We implemented a prototype of Swing State for P4. Given a P4 program, Swing State performs static analysis to compute which states require consistent migration and automatically augments the program to enable the transfer of these states at runtime. Our preliminary results indicate that Swing State is practical in migrating data-plane states at line rate with small overhead.

CCS Concepts
- Networks → Network architectures; Programmable networks; Network management;
*Work performed while interning at ETH Zürich.

Keywords
Network updates; Software-Defined Networking; P4; Stateful programmable data planes.

1. INTRODUCTION
By enabling stateful applications to run directly in the data plane, at line rate, programmable data planes [9, 23, 3, 41, 25, 10, 24] have recently emerged as a promising research area.

Yet, despite making SDNs more powerful, maintaining states in the data plane also calls for new consistent update mechanisms as it prevents traditional update techniques from working, and this, for three main reasons. First, the fact that data-plane states can be updated at line rate—at speeds that can reach Tbps [5]—prevents any software-based controller from consistently moving states from one device to another. Inconsistent migration is a problem for any data-plane application that requires strong-consistency network-wide. Examples of such applications include stateful firewalls tracking dynamic flow characteristics (e.g., low-level TCP states [30]) or anomaly detection applications [22]. Second, even ignoring states dynamism, the exact set of states to be migrated may actually be unknown to the controller, preventing it from performing the migration in the first place. Indeed, the states location in memory can differ from device to device according to runtime factors (e.g., a hash computed on packet headers) that are invisible to the controller. Third, data-plane states can be shared across multiple flows, forcing these flows to be migrated at the same time to avoid inconsistency. Again, the exact flows to migrate can depend on runtime factors that are invisible to the controller.

This work We present Swing State, a general migration framework for stateful data planes. Swing State addresses the above challenges by consistently moving states from one device to another entirely within the data plane, at line rate. The key idea is to have each packet record the state values it reads at the source data plane, carry them to the destination device (through piggybacking), and override the memory locations it reads there. Once the corresponding states in the source...
and the destination devices are synchronized, flows can be migrated using any existing network update techniques such as Swing State is generic and enables to consistently shift data-plane states pertaining to any P4 program, without human intervention.

Swing State achieves consistent data-plane state migration in three consecutive steps. First, prior to deploying a P4 program, Swing State automatically analyzes it to figure out: (i) which states require live migration (because they cannot be safely reconstructed from the traffic); and (ii) which flow headers can update them at runtime. Second, Swing State augments the P4 program to enable the live migration of these states. Third, upon a state migration request pertaining to a set of flows, Swing State configures the source switch to piggyback the relevant state values onto the corresponding traffic. The destination switch then decapsulates these values and overrides its own states accordingly. Once the states are synchronized, Swing State lets the source temporarily mirror the relevant traffic to the destination and notifies the controller that it can safely reroute the flows.

Novelty While consistent network updates has been the topic of extensive research (e.g., [11]), we are not aware of any technique ensuring per-packet consistency in the presence of data-plane states. Also, with respect to consistent state migration initiatives in the context of Network Function Virtualization [13], middleboxes [25], or network controllers [11, 27, 53], the key novelty of Swing State is that it works at line rate, over hardware-based data planes. In contrast, previous works focused on migrating software-maintained states that are up to orders of magnitude less dynamic.

Contributions To sum up, our main contributions are:

- Swing State, a general state-management framework alongside with a runtime system which enables live state migrations in any P4-enabled network (§2);
- A static analysis algorithm which automatically identifies the states requiring live migration in a P4 program (§3), together with an augmentation procedure to actually support live migration at runtime (§4);
- An efficient state synchronization procedure (§5);
- An implementation of Swing State along with a preliminary evaluation assessing its feasibility (§7).

2. MOTIVATION

In this section, we first explain with a simple example how P4 program leverages data-plane states and why it is hard to shift them around (§2.1). We then describe the core principles behind Swing State (§2.2).

2.1 Background

Stateful P4 data planes In P4-enabled switches, data-plane states (stored in registers) reside in the device’s ingress or egress pipelines and are maintained by actions. P4 developers construct data-plane functions by defining match+action tables, along with control flows, and header parsers/deparsers.

As an illustration, P4 enables to easily implement a stateful firewall which automatically drops the traffic originating from heavy hitters and load-balances the rest. Figure 1 depicts one possible implementation [2]. It involves two tables connected with a conditional control flow. The first table, set_heavy_hitter_count_table, counts the packets in each TCP flow (m.pktcnt), before caching the result in hh_pktcnt by hashing the packet’s header (see Line 15 in Figure 1). If this count is larger than HH_THRESHOLD, the packet goes to the drop_heavy_hitter_table where it is dropped; otherwise, it goes to the reset ingress pipelines where it is load-balanced according to the flowlet it belongs to.

Update scenario Suppose that switch S3 (Figure 2) runs the aforementioned application and that flows from S1 to S2 (crossing S3) have been flagged as heavy. Now consider that we need to reboot S3 (e.g., to perform a firmware update). To avoid impacting the traffic, we want to move flows away from S3 to S4. Yet, simply
shifts flows from S3 to S4 (e.g. using \[28\]) would cause the runtime states stored in \(hh\_pktcnt\) to be lost, allowing traffic that should be dropped to go through.

This example sheds light on two fundamental questions regarding consistent data-plane states migration:

**What to migrate?** Not all states require consistent migrations: some functions can automatically recover their states from live traffic. This is for instance the case for our implementation of flowlet detection (Figure 3) which records each flow’s last reference time and current flowlet ID in register \(lasttime\) and \(flowlet\_id\). Migrating these states is not necessary as they can be reconstructed nearly immediately at the destination.

**How to migrate?** The simplest way to migrate states is to request the control plane to export the states from the source device to the destination device (e.g., similarly to \[13\]).

Unfortunately, simply migrating states via the control plane does not work because of: (i) the speed at which states can be updated (Tbps in the new generation of programmable data-planes \[8\]); and (ii) the flexible support of state references which makes it hard, if not impossible, to infer the exact states location at runtime.

The latter problem results from the fact that many applications (e.g. \[1, 13, 22, 28\]) reference per-flow states using a hash of the corresponding packet headers. As different switches might use different set of inputs for hashing or have different capacity/size for the register arrays, the resulting state location can end up being device-specific. For instance, the destination switch might employ a less-specific input for hash calculation (e.g., based on 4 tuples instead of 5) and have a larger \(\text{REG\_SIZE}\), in which case each flow’s state location (e.g., Line 40, Figure 4) would shift. Even worse, the fact that P4 supports the use of runtime data (e.g., action parameters) as input to the hash functions can make it impossible for the control plane to infer the exact reference to use for accessing each state at compilation time.

2.2 Overview

To support consistent and live network updates in stateful data planes, we propose **Swing State**, a general state-management framework and runtime system offering one main primitive: \texttt{moveStates} (Figure 4). At its core, Swing State adopts the novel idea of automatically identifying the states requiring consistent migration then letting each packet/flow itself move the state values it has read by leveraging the programmability of data plane. With Swing State, devices can perform live migrations of data-plane states without freezing traffic nor the rule updates made by the control plane.

Swing State capability of moving states at runtime means that developers can write P4 applications without having to care about migration. At deploy time, Swing State Analyzer analyses their P4 program to infer which states: (i) require migrations; and (ii) are shared between multiple flows meaning they should be treated as a whole. Based on this analysis, Swing State Mod-
ifier automatically augments the program to support live state migrations. By using the augmented program, reconfigurable devices automatically support consistent and live migration for data-plane states.

Once the SDN controller wants to migrate a set of flows \( f \) from one device to another, the State Manager first checks whether this migration is safe based on its state analysis. In case \( f \) shares critical states with others, or it uses device-specific states (see below), the State Manager raises an alert along with remarks. Otherwise, the State Manager configures the source and destination devices to migrate data-plane states. Once all required states have been migrated, the State Manager notifies the controller that it can update \( f \)'s paths safely.

### 3. Static Analysis

In this section, we describe how Swing State analyzes P4 programs to identify state types \((\{3, 4, 11\})\) along with the corresponding flow spaces that use them \((\{8, 9, 12\})\).

#### 3.1 State taxonomy

We classify P4 data-plane states along two dimensions: (i) their usage (soft vs hard); and (ii) whether their values are location-dependent or not.

**Property 3.1** (Soft state vs Hard state). A P4 state is soft if its value is computed from, or maintained depending on, random variables, such as the time stamps of events triggered by packets (e.g., arrive, enqueue, dequeue, or leave), the occupancies of queues, meter values, etc. Otherwise, the state is considered as hard.

Soft states are typically used for optimization purposes in congestion control algorithms, scheduling, and active queue management \((28)\). As the values of soft states are essentially random, the data-plane functions tolerate inconsistency by design. As an example, in Figure 1, the states stored in `lasttime` and `flowlet_id` are soft as they depend on packet arrival times.

In contrast, hard states are maintained deterministically and explicitly (e.g., according to a state machine) and cannot easily be recovered from live traffic. As an example, the packet count stored in `hh_pktcnt` used by heavy-hitter detection (Figure 1) is hard. Other examples include security applications, such as stateful firewalls or anomaly detection, whose state machines depend on few key observations (e.g., TCP state tracking \((11)\)) that only happen once in the lifetime of a flow. The mapping between virtual IPs (VIPs) and direct IPs (DIPs) found in any network load-balancer \((11, 12)\) is another example of hard state which is usually set at the beginning of the connection.

From an update viewpoint (see Table 1), only hard states require to be migrated as soft states can directly be reconstructed at the new location of the flow, at the price of a slightly less efficient (but still, correct) network.

**Property 3.2** (Location dependency). A P4 state is location-dependent if its value is device-specific, such as port id, local time stamps of events triggered by packets, or the current occupancy of a queue.

Some hard states only make sense locally and/or depend on the network topology. To avoid correctness issues, these hard states therefore need to be “translated” to the corresponding state representation used at the target device. For instance, consider a switch running a stateful application which builds a list of MAC addresses authorized to send traffic on each physical port. Shifting flows crossing this switch to another one requires to move the corresponding decisions, while adapting the references to the physical ports to corresponding ones on the target switch.

**Swing State** requires the developers to write specific transfer functions \((27)\) to migrate hard and location-dependent states. While writing transfer functions requires detailed knowledge of the application and the topology, most of the location-dependent states encountered in practice are soft and therefore require no migration (nor transfer functions). Indeed, none of the stateful P4 programs we analyzed \((3, 1, 11, 18, 15, 20, 22, 28)\) required transfer functions.

#### 3.2 Flow-space dependencies

If multiple flows read and write the same state, they should be migrated together, as doing otherwise could cause inconsistent forwarding. **Swing State** needs therefore to be aware of the flow space using each state.

Given a P4 state, the flow space that uses it is given by all the expressions used in the control flow selection, action lookup, and the index calculation. Unfortunately, P4 also supports state indexes to be computed from runtime factors such as any other register values and action parameters (e.g., \((18)\)), which makes it impossible to precisely figure out at compilation time which set of packets share any given state.

To address this problem, **Swing State** considers a generalized flow space when runtime inputs are used (similarly to \((17)\)). Specifically, **Swing State** abstracts away runtime inputs and only considers the subpart directly parsed from the packet and employed by the state’s reference/index calculation. As an example, the precise input space of `lasttime`’s reference (Figure 1) implicitly involves `m.pkt <= HH_THRESHOLD`, the expression employed by the control flow (Figure 1). Overlooking it to treat values in `lasttime` as per-flow states, indexed by the unmodified 5-tuple, results in a conservative answer.

#### Table 1: Only hard states require to be consistently migrated as soft states can directly be reconstructed at the target device. Location-dependent states further required to be transformed to ensure compatibility.

<table>
<thead>
<tr>
<th>Location</th>
<th>Dependent</th>
<th>Independent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft</td>
<td>No migration (e.g. (3))</td>
<td>No migration (e.g. (3))</td>
</tr>
<tr>
<td>Hard</td>
<td>Data-plane migration with transfer function (e.g. (1, 11, 13, 21, 21))</td>
<td>Direct migration</td>
</tr>
</tbody>
</table>


4. MAKING P4 STATES “SWINGABLE”

In this section, we describe how Swing State augments P4 programs to enable runtime migration. We focus on migrating hard states involved in the ingress pipeline (where most of the processing lies).

4.1 Forwarding modes

Swing State achieves live state migration by introducing four types of local forwarding modes:

1. **normal_fwd**: the default mode in which the data plane forwards the packet normally.

2. **state_pickup**: a mode appearing only at a migration's source device in which the data plane: (i) forwards the packet normally, while recording the used state values; and (ii) makes a clone of the original packet, encapsulates it with the recorded values, then tunnels it to the destination device.

3. **state_putdown**: a mode appearing only at a migration's destination in which the data plane: (i) decapsulates the packet to get its original header and state values; (ii) processes the original header normally while overriding each state before reading it; and (iii) drops the processed packet at the end.

4. **mirror_fwd**: a mode having different meanings for a migration's source and destination devices. In this mode, the source device forwards the packet normally while tunneling a clone to the destination, while the destination processes the decapsulated packet normally and drops.

Figure 5 illustrates how Swing State synchronizes the data-plane states at the source and destination devices by changing the forwarding mode assigned to packets from normal_fwd to state_pickup to mirror_fwd. We describe the process in more details in 4.2.

4.2 Program augmentations

Swing State automatically augments P4 programs to support these forwarding modes (Figure 6). The key insight is to let each (stateful) action itself record and override the read state values for each packet. To achieve this, Swing State inserts code snippets just before each read access to hard states. For instance, consider the stateful action set_hh_count in Figure 7. hh_pktcnt tracks flows’ packet counts where that number of the flow in process is identified in process is hh_pktcnt[m.flow_id]. Swing State generates metadata field _SS_m.hh_pktcnt_0 for the reading of hh_pktcnt[m.flow_id] at Line 5 (Figure 7).

If a packet is in the mode of state_pickup, the augmented set_hh_count would cache the observed value (Line 7) and set metadata _SS_m.stateful to 1 (Line 8), indicating the packet being processed is using states. Then the augmented pipelines would: (i) make a clone of the original packet (3, Figure 8); (ii) encapsulate the clone with this cached state value; and (iii) tunnel it to the target data plane (4, Figure 8). In case of state_putdown, the augmented parser and ingress pipeline decapsulate the received clone and cache the state values in _SS_m.hh_pktcnt_0 (1, Figure 9) which is then used by set_hh_count to overwrite the value of hh_pktcnt[m.flow_id] (Line 4, Figure 9); finally, this clone packet gets dropped (3, Figure 9).

5. MANAGING STATE MIGRATION

Migrating states from switch A to B for a given flow space f includes 4 steps.

1. **Configure B to accept states destined to it.** Swing State employs specific tags (i.e., tid) to identify concurrent migration tasks. Upon receiving an encapsulated packet, B’s ingress pipeline, i.e., 1 in Figure 10.
Figure 7: Swing State Modifier inserts Line 3 (in pink) to enable state overrides, and inserts Line 4 and 7 (in lime) to enable state records, by using ternary operators [11].

first checks whether this packet carries state values. If so, \( \text{if} \) decapsulates the packet to get the original header, caches the carried state values in pre-defined metadata fields (e.g., _SS.m.hh_pktcnt_0), and sets this packet’s work mode as state_putdown. The actions then overwrite the states read by this packet.

2. **Activate A to emit f’s states.** To let A’s data plane record f’s states, Swing State inserts match+action rules into \( \text{f} \), so that f’s packets would be marked as state_pickup. If some state values have been recorded during \( \text{f} \) (i.e., _SS.m.stateful=1), \( \text{f} \) will clone this packet to egress pipeline via primitive action clone_i2e, then \( \text{f} \) encapsulates this clone with the recorded state values and delivers them to B via tunnel tid. From now on, all the state values used by \( \text{f} \) would be automatically synchronized/mirrored to its target data plane, B.

3. **Wait for incoming packets.** As states are piggybacked on traffic, Swing State waits for matching traffic to trigger the migration process.

4. **Activate mirror_fwd for f on A.** After Step 3, all f’s state values in A and B have been synchronized. Swing State then configures A’s \( \text{f} \) to set f’s work mode as mirror_fwd: f’s incoming packets will be mirrored to B. Then B processes them as normal and drops.

After Step 4, all states involving f have been migrated and the flow can safely be moved from A to B.

6. **PRELIMINARY EVALUATION**

We have successfully used Swing State to analyze and augment the P4 application shown in Figure 4.

Implementation As the API of a P4 data plane is automatically generated from its code, managing the actual state migration (using the match+action rules described in \( \text{f} \), \( \text{g} \), and \( \text{h} \)) is relatively straightforward. Also, since the two current versions of P4, 1.0.2 and 1.1.0, are not syntactically incompatible, both the analysis and augmentations are performed on the JSON-formed Intermediate Representations (IR) outputted by the P4 front-end compiler and which is designed to be consistent across different versions [11].

Figure 8: Without Swing State, the flow’s packet counts used by heavy-hitter firewall would get lost (i.e., values in hh_pktcnt), resulting in allowing packets that should be dropped (the threshold of dropping is 100).

**Case study** We check whether the augmented P4 application supports consistent network updates by reproducing the example of Figure 4 (moving flows from switch S3 to S4). We set the threshold of heavy hitter to 100 and let S1 send packets to S2 via a TCP connection. Figure 5 shows how the number of received packets (at S2) changes when the network is updated with and without Swing State. With Swing State, the values of packet counts get migrated correctly; thus, the stateful firewall works perfectly as no update happened.

**Limitations & future work** While promising, the current (preliminary) version of Swing State is limited and requires future work. First, Swing State is not designed to deal with: (i) packet re-ordering and loss; and (ii) inconsistent hash collisions between different hash implementations. In rare cases, these issues might lead to inconsistent migrations. Second, Swing State does not currently support state merging operations should the source and the target switch have states in common. Finally, our current implementation of Swing State can mirror multiple times the same state value, resulting in wasted bandwidth. A possible solution here is to filter duplicates at the source using a bloom filter.

7. **CONCLUSION**

This paper introduced Swing State, a general framework for migrating data-plane states for programmable switches. By directly piggybacking state values on traffic, Swing State migrates data-plane states without freezing the traffic nor control-plane updates. We implemented a Swing State prototype and showed that it can automatically analyze and augment P4 programs as well as successfully perform a live migration of the states pertaining to a heavy-hitter firewall.

**Acknowledgments**

We would like to thank our shepherd, Jeongkeun Lee, for his invaluable help in improving the paper and the anonymous reviewers for their constructive feedback. Shouxi Luo was supported by the China Scholarship Council during his stay at ETH Zürich.

```c
action set_hh_count() {
    modify_field_with_hash_based_offset(m.flow_id, 0, 13_hash, REG_SIZE);
    hh_pktcnt[m.flow_id] = (_SS.m.pkt_wmode == state_putdown) ?
        _SS.m.hh_pktcnt_0 : hh_pktcnt[m.flow_id];
    m.pktcnt = hh_pktcnt[m.flow_id] + 1;
    hh_pktcnt[m.flow_id] = m.pktcnt;
    _SS.m.stateful = (_SS.m.pkt_wmode == state_pickup) ? 1 : _SS.m.stateful;
}
```
8. REFERENCES


