Blueswitch: Enabling Provably Consistent Configuration of Network Switches *

Jong Hun Han† Prashanth Mundkur‡ Charalampos Rotsos* Gianni Antichi†
Nirav Dave‡ Andrew W. Moore† Peter G. Neumann‡

†University of Cambridge
{firstname.lastname}@cl.cam.ac.uk

‡SRI International
{mundkur, ndave, neumann}@csl.sri.com

*University of Lancaster
c.rotsos@lancaster.ac.uk

ABSTRACT

Previous research on consistent updates for distributed network configurations has focused on solutions for centralized network-configuration controllers. However, such work does not address the complexity of modern switch datapaths. Modern commodity switches expose opaque configuration mechanisms, with minimal guarantees for datapath consistency and with unclear configuration semantics. Furthermore, would-be solutions for distributed consistent updates must take into account the configuration guarantees provided by each individual switch – plus the compositional problems of distributed control and multi-switch configurations that considerably transcend the single-switch problems. In this paper, we focus on the behavior of individual switches, and demonstrate that even simple rule updates result in inconsistent packet switching in multi-table datapaths. We demonstrate that consistent configuration updates require guarantees of strong switch-level atomicity from both hardware and software layers of switches – even in a single switch. In short, the multiple-switch problems cannot be reasonably approached until single-switch consistency can be resolved.

We present a hardware design that supports a transactional configuration mechanism, and provides *packet-consistent* configuration: all packets traversing the datapath will encounter either the old configuration or the new one, and never an inconsistent mix of the two. Unlike previous work, our design does not require modifications to network packets. We precisely specify the hardware-software protocol for switch configuration; this enables us to prove the correctness of the design, and to provide well-specified invariants that the software driver must maintain for correctness. We implement our prototype switch design using the NetFPGA-10G hardware platform, and evaluate our prototype against commercial off-the-shelf switches.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)

*This work was jointly supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article/presentation are those of the author/ presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the support of the UK EPSRC for contributing to parts of our work, through grant EP/D076803/1.

Keywords

OpenFlow; switch configuration; atomic transactions; OpenFlow bundles; NetFPGA

1. INTRODUCTION

Network administration involves the specification of policies for traffic handling, including the routing, shaping and dropping of network traffic. These policies translate into detailed flow-table configurations of individual switches. Formal approaches to such specifications and translations [1–5] provide mechanisms and algorithms to safely handle updates by managing the transition from one configuration state to the next. Reitblatt *et al.* [1, 2] formally define *packet consistency* to specify desirable properties that configuration updates should maintain. A packet-consistent configuration update requires that *every* packet traversing the network be processed using either entirely the previous configuration, or the updated one, but never a mixture of the two.

Consider a single network-administration domain – with a collection of network elements subject to a common set of policies. An operational requirement would be the transition of this network from an old configuration to a new configuration. To achieve consistency, switches within the network must simultaneously handle both old and new versions of the desired configuration at the same time.

One approach is to deal with these multiple versions with no changes to existing network hardware. Previous work [1, 2, 6] has adopted this approach, and proposed embedding appropriate configuration version numbers into existing fields (*e.g.,* VLAN tags or MPLS labels) of the Ethernet frames of packets flowing in the network. These version number tags indicate a particular version of the network configuration, and would be injected into packet frames when the packets enter the administrative domain and then stripped when they leave.

The reconfiguration of a new policy into any switch is done using a sequence of configuration commands issued to the switch configuration interface. This requires the use of the programming contract provided by the low-level implementation of this configuration interface. However, in practice, these interfaces provide little or no guarantee regarding the order in which the configuration commands are processed and then manifested in the datapath [5].

As a result, in OpenFlow contexts, the entity computing a configuration update (typically an OpenFlow controller) often needs to resort to a pessimistic (worst-case) approach, *e.g.,* the use of the OpenFlow *barrier* protocol. On the receipt of a *BarrierRequest* from a controller, the switch must respond with a *BarrierReply* when the preceding configuration commands have been completely processed. Even then, the switch is not constrained to process and install the individual commands within a transaction in an ordered

and timely manner; in fact, as we show, commercial switches often take advantage of this lack of constraint, presumably for performance reasons. In the worst case, a barrier would be needed for each individual configuration command to enforce ordering in a command sequence.

However, a barrier transaction per individual command is still insufficient in the case of modern switches. The design of modern commercial OpenFlow switches (*e.g.,* Broadcom OFDPA [7]) involves pipelines of multiple specialized flow tables. A single policy modification frequently translates into multiple potentially parallel and asynchronous updates across different flow tables in a single switch. The ordering of these updates with respect to the flow of packets within the datapath can give rise to anomalies in the packet flow, leading to possible violations of the desired switching policy. In such cases, even if every single configuration command was sequenced using a barrier transaction, there would still be no guarantee that the updates to the multiple tables are manifested in a packet-consistent manner in the datapath. Such inconsistencies can easily lead to packet loss, with subsequent performance impact. They also raise serious security concerns, since the resulting misrouting of packets can cause violations of network security policies. An attacker capable of triggering reactive configuration updates (*e.g.,* by intrusion-prevention systems) can cause repeated packet misrouting, leading to traffic (and information) leakage.

In this paper, we argue that a crucial building block for packet-consistent updates in an OpenFlow network with multiple switches is the provision of *packet consistency in the internal datapath of each switch*. We emphasize this point by experimentally demonstrating consistency violations in commercially available switches, in various scenarios. Such inconsistencies at the individual switch level can violate any consistency guarantees provided by proposed solutions to the distributed network update problem. We speculate that the relevance of such problems motivated the introduction of the *Bundle* feature in version 1.4 of the OpenFlow standard. *Bundles* support atomic updates to a switch configuration; however, no commercial switches that we know of currently support this feature.

These problems provide the motivation for our design of a switch datapath and its configuration interface – which we present in detail. This design provides one possible implementation of the OpenFlow *Bundle* feature. We prove that the design provides the desired consistency in its configuration interface, relying crucially on invariants implemented in hardware as well as invariants that need to be preserved by the software driver of the switch.

The contributions of this paper are:

- A demonstration of the lack of guarantees in the configuration interfaces of commercially available switches, and the resulting violations in packet-routing policies.
- A precisely specified hardware-level transactional configuration mechanism for multi-table pipelined switch datapaths that achieves packet consistency, along with a proof of its correctness.
- An evaluation of our OpenFlow switch prototype (*Blueswitch*) that implements this mechanism on the NetFPGA-10G platform. We believe this is the first pipelined multi-table OpenFlow switch on an FPGA, which is also the first switch to have a provably correct configuration interface to enforce packet consistency in its datapath. We plan to contribute this implementation as an open-source project in the NetFPGA ecosystem.

Although our focus in this work is on the update consistency of *individual* switches and the implications for the problem of distributed consistent updates, we are not proposing novel solutions to the distributed consistency problem. Instead, we argue that solutions to the broader problem must rely on individually consistent switches; indeed, previous solutions implicitly assume such behavior. These solutions would work as intended, *provided* every switch ensured consistency during its individual updates – and provided the compositional problems of multiple controllers and multiple switches were properly addressed.

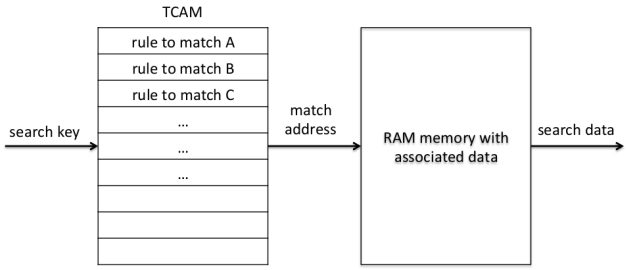


Figure 1: The two memories used in switch datapaths: (i) TCAMs for flow-match definitions, and (ii) RAMs for forwarding actions.

In the rest of this paper, we first provide in §2 some background on switch architectures and their configuration, and illustrate the impact on policy enforcement of the configuration interfaces of commercially available switches. The packet misrouting caused by the lack of guarantees provided by these interfaces motivates our presentation of a detailed architecture of Blueswitch in §3. After describing its datapath in §3.1, we detail the hardware and software interface for switch configuration in §3.2. We provide a proof of how it maintains consistency as well as the constraints imposed on the protocol at the hardware-software interface. In §4, we describe the implementation of Blueswitch on the NetFPGA platform, and evaluate its performance. We discuss the design and its evaluation in §5, and then conclude after a review of related work.

2. MOTIVATION

This section motivates the proposed switch design by presenting the design of lookup memories used in a modern OpenFlow switch (§2.1) and the challenges in maintaining policy consistency in such architectures (§2.2). We focus our analysis on OpenFlow-enabled switches, which implement the required data-plane functionality in a single ASIC.

2.1 Datapath Memory and its Configuration

The forwarding functionality of the packet-processing pipeline in modern OpenFlow switches relies on a fast-lookup abstraction with support for extensive wildcard matching and reconfigurability. The lookup abstraction, called a *flow table* in the OpenFlow specifications, stores the forwarding policy and state. Rule entries in the flow table are logically separated into two units: (a) the flow-match definitions, which define the matching value for each field of the lookup tuple of an input packet, and (b) the action list, which contains the operations that need to be performed on each matched packet (such as sending it out on a specific physical port).

Hardware switches predominantly implement flow tables using two memory subsystems embedded in the ASIC. The flow-match definitions are stored in one or more Ternary Content-Addressable Memory (TCAM) modules. TCAMs are optimized for fast lookups, supporting $O(1)$ lookup complexity for any match, irrespective of the lookup key-width or the number of stored flow-match definitions. The output of a TCAM lookup is used to index into a regular RAM memory module, which stores the flow actions and statistics [8, 9]. Figure 1 shows the interaction between TCAMs and

RAM memory in high performance switching engines. In routers, for instance, an IP address is used as the input key to the TCAM, while the output value provides the RAM address that stores the forwarding port and the next-hop IP address. OpenFlow switches often use multiple TCAM flow tables to store the configured rules.

The rules in the flow tables are managed by the control software of the switch, which runs on the switch co-processor. The control framework translates high-level OpenFlow command from the controller into equivalent flow-table modifications. The flow tables expose a hardware configuration interface to the co-processor (typically a set of memory-mapped register ports accessed over a dedicated channel such as PCIe) that allows the control software to manipulate the flow-table rules.

When a rule entry is inserted into or deleted from a flow table, both the flow-table TCAM and its associated RAM must be updated. During configuration updates inconsistencies may arise, since the two memory modules are weakly associated and possess different update latencies. Ensuring this consistency is complicated – because the update latency of a single TCAM entry is not constant and depends on the memory design and on the number and structure of the entries already installed, whereas the update latency of a RAM entry is constant and depends solely on the memory type.

When a datapath is implemented using multiple flow tables, a rule update often requires the modification of the TCAM and RAM memories of each flow table. This exacerbates the problem of maintaining consistency between the memory modules of all the flow tables, especially while packets continuously traverse the datapath, due to the myriad race conditions involved in updating multiple TCAM and RAM memories. In addition, the number and latency of the read/write port IO operations that the control software needs to perform to effect a single rule update, across all the flow tables, depends on the speed and latency of the interface between the control and switching processors.

One implementation choice to guarantee atomicity during flow-table updates by the control software is to first drain and then block the processing pipeline before installing the update, thereby avoiding the possibility of forwarding inconsistencies. However, this option significantly increases both packet-forwarding latencies and latency variability during updates. Another choice is to use a double-buffered flow-table pipeline that uses shadow tables where an updated configuration is written, along with an efficient commit operation that swaps the active and shadow tables. This avoids blocking the processing pipeline, and hence reduces packet latencies through the switch – but at a possibly increased memory cost.

2.2 Policy Enforcement during Configuration Updates

The above discussion on the memory architecture in a switch datapath illustrates the complexity in maintaining consistency during configuration updates. Most switches do not provide any guarantees on the installation order and latency of a sequence of updates; as a result, such updates need to be carefully sequenced to not violate policy [5].

Network administrators implement policies by configuring the rule entries in the switch flow tables. Correct enforcement of network routing policies is critical in network administration. Policy enforcement requires not only that each configuration implements the desired policy, but also that policy violations do not occur during configuration updates.

A common approach to guaranteeing the ordering of a sequence of rule modifications in OpenFlow architectures employs *Barrier* messages, as described in §1. However, commercial switches show a considerable diversity in their implementations of the *Barrier* protocol, and their use does not always effectively provide the guarantees sought – for several reasons. Firstly, various switches do not enforce such barriers in a manner consistent with specifications. Secondly, using the *Barrier* protocol introduces significant network round-trip latencies during large policy updates – which consequently affects the overall performance of the network. Thirdly, because the forwarding control logic is distributed between the software and hardware layers of the switch, the provision of effective guarantees requires the tight coordination between the two layers. In the absence of this coordination, weak temporal synchronization between layers is exposed, causing transient policy inconsistencies with potentially significant security implications.

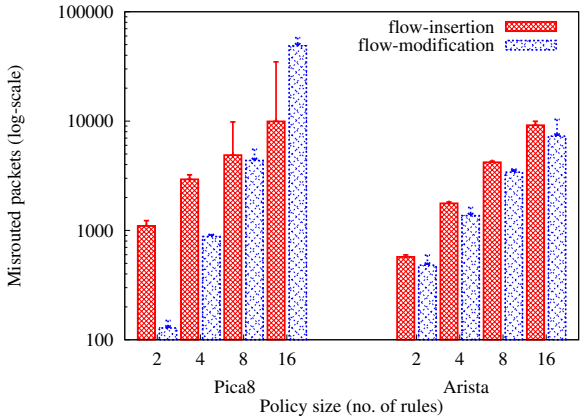


Figure 2: Number of packets misrouted during policy updates for Arista 7050S and Pica8 P3922. Both switches exhibit a significant number of packets misrouted (y-axis) during small policy reconfigurations; the number of reconfigurations is shown along the x-axis, pairwise for flow insertion and flow modification. The evaluation for Blueswitch is presented in §4.4.

To demonstrate these problems, we conducted experiments employing a simple measurement setup described in detail in §4.2. We compared Blueswitch with two 10 GbE OpenFlow-enabled commercial switches, the Arista 7050S [10] and Pica8 P3922 [11], and connected them directly to a host capable of high-precision traffic generation and capture. Traffic was generated at an aggregate 2 Gbps rate of small-sized (150-byte) packets, and *FlowMod* operations were used to evaluate the impact of policy reconfiguration on data-plane performance. During each experiment, we initialized the switch with a set of rules that forwarded traffic to a set of destination IP addresses via a specific output port. After a sufficient warm-up period, during which we generated traffic targeting the installed rules, we changed the output port using *FlowMod* messages, and measured the latency and impact of the policy update on the data and control planes.

The *flow-insertion latency* experiment initialized the switch with a single low-priority wildcard flow matching the destination IP address set. The output port for each destination IP was then changed by inserting new higher priority rules for each destination. The *flow-modification latency* experiment initialized the flow table with a set of destination-specific rules, one for each destination IP. Then, a corresponding set of rules modified each output port, which effectively modified only the action list of the initial rule set.

During each experimental run, we sent a *BarrierRequest* message after each batch of *FlowMod*s, and used the *BarrierReply* to measure the processing latency of the switch. The two experiments provide two modification scenarios; the first requires the agent to update both the TCAM table as well as the associated RAM, while

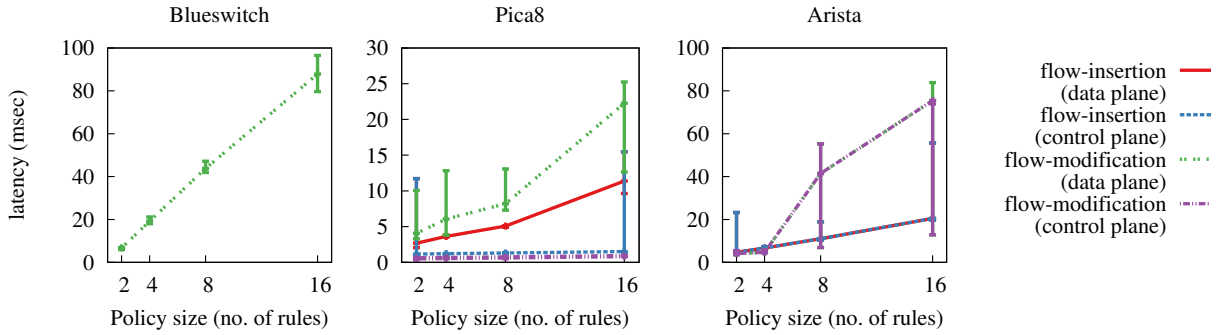


Figure 3: Flow-table reconfiguration latency for the Blueswitch, Pica8 P3922 and Arista 7050S switches. Reconfiguration latency is measured using Barrier messages (control plane) and time-stamped data-plane packets (data plane).

(theoretically) the second modifies only the RAM.

Figure 2 presents the number of misrouted packets detected during configuration updates for the two switches. We define a packet in a flow as *misrouted* if it is forwarded through the output port of the old configuration after any packet has been received from the output port of the new configuration. The results show that even for small policy updates, production switches exhibit significant packet misrouting. For example, when only two rules were modified, the Arista switch misrouted 130 packets, while the Pica8 switch exhibited routing inconsistency for 480 packets. The number of misrouted packets increases roughly proportional to the size of the configuration change, while flow *insertions* in most cases incurred more policy violations than flow *modifications* (although large flow modifications in the Arista switch differed in this respect).

We also explored the implementation of the *Barrier* protocol in these two switches, and contrasted them with Blueswitch. Figure 3 shows the latencies measured for applying the configuration updates described above. We measured two different latencies: (i) the *data-plane latency* (which measures the time between the transmission of the new configuration and the time that at least a single packet was received on the new port for each flow), and (ii) the *control-plane latency* (which measures the time between the transmission of the new configuration and the receipt of a *BarrierReply*).

We observed significant differences in the other two switches, both in the semantics of their *Barrier* implementations and their performance. The Pica8 switch responded with a *BarrierReply* as soon as it had received and installed all rule updates in the control software stack of the switch. This implies that the switch implementation considers a configuration update to have occurred when the *FlowMod* is received and installed in software, but independent of installation in the hardware datapath. This behavior is inherited from the OpenVSwitch architecture, which uses multiple rule caches and adaptive rule installation to improve performance. Furthermore, a close inspection in the installation order of new rules through data-plane measurements revealed that rule installation in Pica8 does not follow the ordering of *FlowMod*s in the update sequence.

In contrast, the Arista switch exhibited a more consistent behavior. Its *BarrierReply* latencies corresponded to the observed data-plane latencies, while data-plane measurements showed that the installation of new rules preserved the ordering of the *FlowMod*s in the update sequence. However, this consistent behavior incurred a performance penalty; the rule installation latency was an order of magnitude higher than that of the Pica8.

Our experiments employed simple reconfiguration scenarios and focused on inconsistencies occurring within single flow-table switch setups. Even in such simple scenarios, production switches exhibited significant policy inconsistencies, while the observed atomicity semantics were diverse. In the case of more complex switch datapaths that are composed of processing pipelines with multiple tables (as proposed in recent OpenFlow specifications), we anticipate that such inconsistencies will be even more prevalent and more difficult to mitigate.

The various proposed solutions for consistent distributed network update (discussed in §1) need to send multiple *FlowMod*s to each switch affected by the update. The installation of any such *FlowMod* in any switch could cause the packet switching inconsistencies we describe here, negating the consistency guarantees promised by such solutions. However, these proposed solutions would work as intended *if* each participating switch offered a consistent update semantics for every *FlowMod* it installed into its datapath.

3. BLUESWITCH ARCHITECTURE

In this section we present the system architecture for Blueswitch, a multi-table switch design with support for a packet-consistent configuration interface. The primary design goal is to *provably* avoid the misrouting issues encountered in the previous section. We take into account the requirements on the software driver to ensure this goal.

Our design is implemented using a hardware-switched data plane configured with a software driver and switch agent. The software agent implements the OpenFlow protocol and uses the driver to control, configure, and monitor the switch hardware. We first describe the architecture of a generic pipelined multi-table switch to provide a concrete context for the description of the transactional configuration scheme. We then present and prove the correctness of the scheme in the context of a datapath comprised of a linear pipeline. This is compatible with the logical datapath model used by later versions of the OpenFlow standard, and provides an implementation of the *Bundle* feature of OpenFlow 1.4. In §5, we discuss how to extend this scheme and support complex pipelines with forks and joins.

3.1 Multi-Table Datapath Architecture

Datapath Structure

Figure 4 shows the architecture of the datapath. The physical ports supply input packets to the datapath via their associated physical port interfaces. On every packet received at a physical port, the interface inserts the port identifier into the packet metadata. Packets leave the output ports of the switch via a crossbar interconnect de-

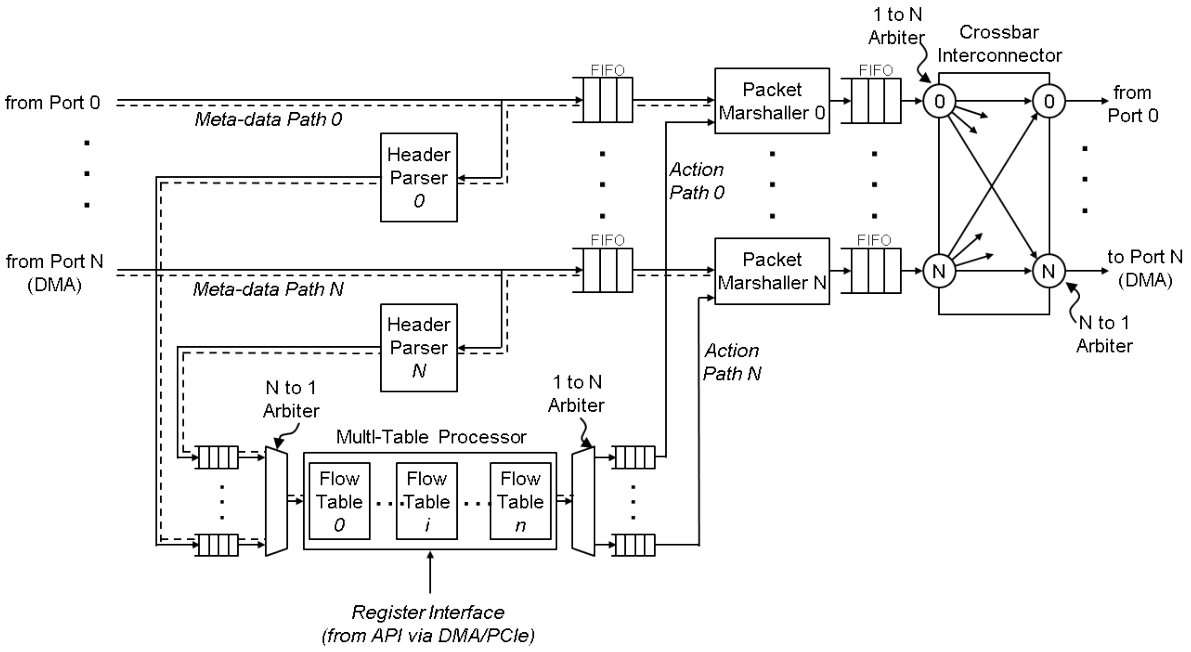


Figure 4: The pipelined datapath. Broken (dashed) lines indicate the flow of metadata attached to packets.

signed with 1-to-N and N-to-1 arbitration modules. The crossbar interconnect routes processed packets to the physical output ports according to the metadata updated by the match processor.

Each incoming packet with metadata is stored in port-specific FIFOs and is parsed in parallel by per-port header parsers. Header fields and metadata parsed by each header parser are serialized through an N-to-1 arbiter into the pipeline. A single processing pipeline is used to save logic costs in the flow-table processor. The flow-table pipeline sends the matched output actions to per-port packet marshallers via an 1-to-N arbiter. The packet marshallers update or modify the header and metadata of the packets stored in the FIFOs to apply the specified output actions, and send the marshalled packets to the output crossbar interconnect.

One important field inserted into the metadata for each packet is V_p, the configuration version of the datapath; its use is described in the following subsection on Flow-Table Structure.

In addition to physical ports, the switch design contains an input port and an output port used to communicate with a host over DMA/PCIe. These two ports allow packet interception and injection for experimental measurement purposes, as described in §4. Switch configuration registers are exposed to control software on the host over PCIe.

Flow-Table Structure

Figure 5 shows the structure of a single flow table. A flow table contains two TCAMs and action processors in a double-buffered configuration. The active TCAM and action processor, T_i, is the one used by the matching logic, while the shadow U_i accumulates new entries and actions from the next incoming configuration update. T_i and U_i swap their active and shadow roles after every transaction commit. S_i stores the transactional state; this sets the active and shadow roles using the table configuration version V_i. A new configuration for the flow table i is set by the control software into U_i using one or more configuration commands. The configuration transaction at i is considered complete when an input packet is received at the table with an updated V_p, at which point the active and shadow memories are swapped by S_i. This process is specified in detail in §3.2.

D_T, D_i and D_A are delay logic elements to coordinate the switchover of the outputs of the TCAMs and action processors; they mediate the programming delays of the TCAM memories.

In the normal case,[1] each flow table i in the pipeline matches selected protocol fields in the packet header against the match-key entries of the TCAM T_i. When an entry matches, the associated actions are performed on the packet and its metadata, as shown in Figure 5. The header fields and metadata are held in a buffer for the duration of the lookup. After the actions are performed, the match results and metadata are passed to the next table $i+1$ in the pipeline.

For our purposes, the most important actions are *Output* (which specifies on which physical port the packet should be output), *Drop* (which indicates that the packet should not be output on any port), and *GotoTable* (which specifies the next table j to process the packet – which may be different from the immediately following table $i+1$ in the pipeline).

Modifications for a single table i are specified in terms of commands sent by the software driver. The commands are either those that modify individual match-action entries, or a distinguished *EndTxn* command. Each table has an associated transactional state S_i that gates the entry of these commands into the shadow U_i. The software driver may send update commands to this table when S_i is *Open*. When an *EndTxn* command is received (it should always be at the end of a sequence of configuration commands for table i), S_i transitions from *Open* to *Primed*. In the *Primed* state, the configuration logic for the table does not accept any more commands into U_i, and incoming packets are processed normally using T_i as long as $V_p = V_i$.

The software driver sends the updates for each table i in this transactional manner, by terminating each update command sequence

[1] That is, when the configuration version V_p of the packet matches V_i of flow table i, as described in §3.2.

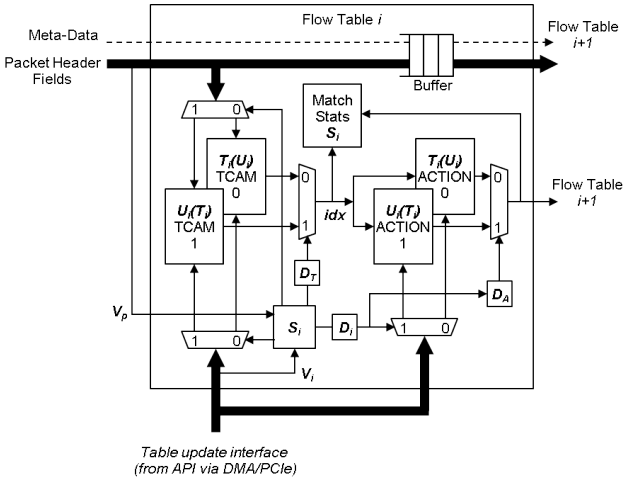


Figure 5: Structure of a flow table.

to i with an *EndTxn*; this results in all the tables ending in the *Primed* state. The driver then increments the global pipeline version number V_p at the entrance of the datapath, which forces new packets entering the datapath to have the updated V_p and causes transaction commits to occur at each table as described below. The hardware configuration logic enforces that V_p can be incremented only if all the flow tables are in a *Primed* state.

To ensure the packet consistency of the pending *Primed* update transactions, each table i compares its configuration version V_i to the V_p in the metadata of the next packet P, that it needs to process. If the table finds $V_p > V_i$, it (*i*) swaps the active T_i and passive U_i, effectively *committing* the update transaction at i, (*ii*) increments V_i to V_p, (*iii*) transitions S_i from *Primed* to *Open* (thus forming an implicit *BeginTxn* for the next transaction), and only then (*iv*) processes packet P. This ensures that P (and subsequent packets at version V_p) will see only the new configuration at this flow table, while any immediately preceding packets see only the previous configuration. This same update processing occurs at each table as the packet P moves along the pipeline.

During normal header processing at each table i, the packet header version V_p is equal to V_i, and the header is processed according to the entries in T_i, unless the metadata specifies otherwise. If a *GotoTable* or *Drop* action is applied to the packet, the metadata is updated appropriately, and the header and metadata are sent to the next table in the pipeline. A table i receiving a packet whose metadata indicates a destination table j where $j > i$, or that the packet is to be dropped, merely passes the packet to the next table without performing any match processing.[2]

On switch initialization, the hardware sets $V_p = V_i = 0$ and $S_i =$ *Open* for each table i.

3.2 Provably Consistent Switch Configuration

To show how this scheme ensures transactional packet-consistent configuration, we explicitly state the invariants enforced by the hardware and required from the software driver. The switch hardware maintains the following invariants:

S1 The tables in the pipeline are ordered linearly, and each packet header flows through every table in order. This packet-header flow is maintained even when *GotoTable* or *Drop* actions are applied to packet headers.

S2 The configuration trigger to increment the pipeline version V_p is enabled only when all the tables are in *Primed* state, thus preventing the software driver from performing an incorrect increment.

S3 No commands are accepted into the shadow U_i when a table i is in the *Primed* state.

S4 The delays D_T, D_i and D_A are synchronized to ensure that, at a configuration commit, the lookup index that is output by a newly active TCAM is used to index into the correct associated action RAM.

S5 The active and shadow memories T_i and U_i are swapped only when S_i is in the *Primed* state, and when the input packet P at the table i has $V_p > V_i$, after which S_i again enters an *Open* state. The role swap is performed before such a P is processed.

In turn, the software driver is required to obey the following invariants:

D1 For each switch configuration change, the update commands sent to each table i need to terminate in an *EndTxn* command. Note that this applies even when no changes need to be made to a table. Hence, each table i needs to be sent at least one command, *EndTxn*, for every configuration change. Otherwise, due to the hardware enforcement of S2, the software would not be able to increment V_p.

D2 The version increment of V_p is triggered after all tables have entered the *Primed* state, and before the start of the processing of the next switch configuration change. This ensures that $V_p \in \{V_i, V_i + 1\}$ for all i at all times.

We can now use these invariants to prove that the scheme implements packet-consistent configuration in the forwarding path.

PROOF. We denote as P the first packet entering the datapath tagged with a newly incremented V_p.

1. S2, D1 and D2 ensure that when V_p is incremented, all flow tables i are in the *Primed* state.
2. S1 ensures that a packet P with an incremented V_p always crosses every table in the pipeline and, from the previous conclusion, finds each table in the *Primed* state.
3. S3 ensures that the shadow U_i of table i remains unmodified from the time V_p is incremented to the time P reaches i.
4. S4 ensures that any packets immediately preceding P tagged with $V_p - 1$ are processed using the previous configuration.
5. S5 ensures the atomicity of the change seen by P.
6. S1 also ensures that when P exits the pipeline, all the flow tables are updated to the new configuration, even if P is marked as *Drop* by an intermediate table.
7. Hence P and all subsequent packets tagged with V_p are processed with the new configuration.

□

Although this proves the safety of the transactional scheme, there is a liveness issue due to the use of the datapath to trigger the update – namely, that if no packets enter the pipeline after the V_p is

[2] *GotoTable* going backward in the pipeline (*i.e.,* $j < i$) is disallowed by the protocol and is typically enforced by configuration software. Our current implementation does not enforce this check in hardware, instead relying on the software driver to ensure it. However, this could easily be done in hardware.

Table 1: Comparison of hardware costs.

Resource	Reference Switch	Reference Router	Blueswitch▽
Slices	18.3K	20.3K	22.9K (25%)
Slice Registers	44.3K	47.2K	57.2K (29%)
LUTs	39.0K	43.1K	43.3K (11%)
LUT-Flip Flop	56.6K	61.7K	72.4K (28%)
No. of TCAMs†	1‡	1	6

†Entry depth of all TCAMs is 32.
‡The reference switch is implemented with a CAM.
▽ Each percentage shown indicates the increase in resource overhead compared with the better of the two reference designs.

incremented, the updates will remain uncommitted in the shadow memories of each table. This would prevent any new configuration updates from being sent by the software driver, because the tables would all remain in the *Primed* state. This is addressed by a hardware inactivity timer at the head of the datapath; this timer is triggered when the software driver performs the increment of V_p. When the timer fires, it forces the commit and role-swap at each table in order. The timer is enabled only when all tables are in the *Primed* state.

We stress that although the implementation of a double-buffered pipeline appears straightforward, its configuration by software requires certain invariants to be preserved by the switch software driver. In this section we have shown precisely what these invariants are, and why they suffice for a correct implementation of a packet-consistent configuration interface.

4. EXPERIMENTAL EVALUATION

This section provides an evaluation of the performance of the Blueswitch design. Specifically, we present a prototype hardware implementation of the proposed update architecture (§4.1) and — using a simple testbed (§ 4.2) — we evaluate the performance of its data-plane (§4.3) and reconfiguration (§4.4).

4.1 Prototype implementation

The prototype Blueswitch was implemented on the NetFPGA platform, which uses the 64-bit AMBA4 AXI-Stream and 32-bit AXI-Lite bus protocols for data transport and register control, respectively. Figure 4 illustrates the implementation architecture, which consists of four Rx and Tx 10GbE-MAC interface modules, input and output arbiters, and the core switch module. Several modules from the NetFPGA-10G platform library were reused. We followed the conventional Xilinx EDK toolchain flow for hardware implementation.

The prototype switch is synthesized at a clock speed of 160MHz, supporting a 10-Gbps full-line rate. The multi-table pipeline shared by all datapaths allows us to implement the consistent configuration architecture without compromising performance. Table 1 summarizes the switch hardware costs on the NetFPGA-10G platform, which are compared to the costs of the reference switch and router [12]. The introduction of the double-buffered TCAM in Blueswitch increases FPGA resources utilization from 11% to 29%, depending on the resource type.

4.2 Experimental setup

To evaluate the prototype implementation, we employed the two-node topology depicted in Figure 6. Both nodes are equipped with a NetFPGA card, a quad-core Intel E5-2687W CPU, and 64 GB

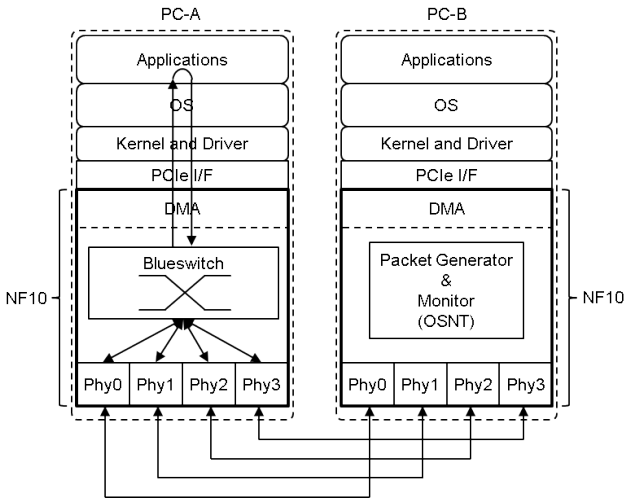


Figure 6: Experimental testbed for our evaluation tests.

of RAM. The NetFPGA card on PC-A was programmed with the Blueswitch implementation, while the NetFPGA card on PC-B was programmed with the OSNT hardware design [13], a high-performance and precision network traffic generator and capturer.

We have recently completed an initial port of OpenVSwitch to Blueswitch [14]. This was completed after the submission of this paper and its acceptance, and it was not used for the experimental results described below.

To configure the forwarding policy in the datapath, we implemented a simple control application running on the host CPU, and used special data-plane packets to trigger policy reconfiguration. Interaction between Blueswitch and the host CPU occurs through a register API as well as through a DMA engine, using a simple kernel-space driver. The register API allows policy reconfiguration, while the DMA engine is exposed in the host OS as a network interface that allows data-plane packet interception and injection. In addition, the design contains a module with a free-running clock-cycle counter, which is used by the control application to measure policy reconfiguration latency in cycles.

To compare our Blueswitch prototype with existing production switches, we use two off-the-shelf 10 GbE switches, the Pica8 P3922 [11] and the Arista 7050S [10]. We performed the evaluations with a set of tests developed using the OFLOPS-Turbo OpenFlow switch evaluation platform [15].

4.3 Data-plane performance

We first examined the impact of the double-buffered TCAM design on data-plane performance. We used the OSNT design to generate traffic at a rate of 5 Gbps with variable packet sizes and configured the measured systems to forward all packets to a specific device port. Using the OSNT high-precision timestamping functionality, we were able to capture all packets and for every packet acquire the transmission and receipt timestamp to compute the forwarding latency. Figure 7 presents a comparison of the median forwarding latency between Blueswitch, Pica8, Arista and the NetFPGA reference switch and router designs. We note that the variance of the latency measurements was negligible across all switches.

The results show that the use of a double-buffered lookup module design does not affect the latency of the switch processing pipeline and can achieve latency comparable to production ASICs. Blueswitch processing latency is surpassed only by the Arista switch, which contains a highly latency-optimized datapath. It is

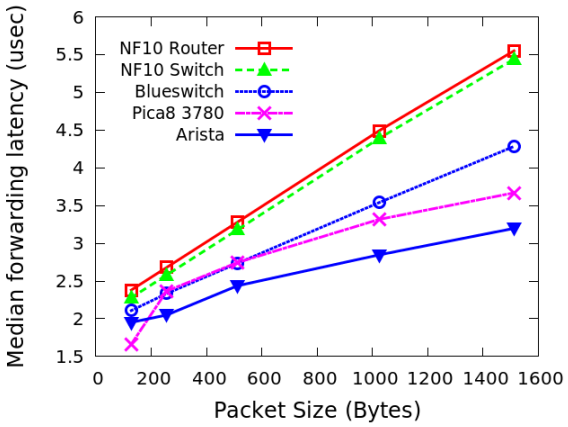


Figure 7: Forwarding latencies for Pica8, Arista, the NetFPGA router and switch designs, and Blueswitch. The Blueswitch latency lies between the production switches and the router.

worth noting that the Blueswitch processing latency is lower than the NetFPGA reference designs. Upon investigation, we found that the interconnect in Figure 4 performed more processing in parallel, thus improving latency. Specifically, the reference NetFPGA designs employed a single arbitration module that also performed data-width conversions, at both input and output ports. This design choice improves programming flexibility, but introduces significant latency in data arbitration.

4.4 Policy reconfiguration

We also analyzed the impact of policy reconfiguration on data-plane performance. Specifically, we address two questions: (i) How are packet latency and routing integrity affected during policy reconfiguration? (ii) How long does the policy reconfiguration take?

To answer the first question, we compare the behavior of the switch using conventional TCAM modules (without the use of the consistent update scheme) against its behavior when the configuration is enforced using the consistent scheme. Our evaluation considered two cases: (a) *flow-modification* tests (which measured the effect of reconfiguring the action lists in the RAM modules of the match tables), and (b) *flow-insertion* tests (which measured the effect of the removal and insertion of new rules in the TCAM modules of the match tables). During each test, we generated data-plane measurement probe packets on two ports of the switch at an aggregate rate of 2 Gbps, and measured the round-trip time of each packet. The measurement probe uses small UDP packets (150B) with destination IP addresses that match the rules of the flow-table in a round-robin manner.

Figures 8a and 8b presents the results of the rule-modification experiment, comparing the data-plane effect of the configuration of conventional TCAMs with our consistent configuration scheme, respectively. This experiment initialized the switch configuration with rules for 12 flows (with destination IP addresses of 192.168.0.2-13). The rules matched the destination IP address of the packets and distributed the incoming measurement traffic, which was sent to the switch port Phy0, between the Phy1 and Phy2 switch ports. After a sufficient warmup period, a policy reconfiguration was applied by the Blueswitch control application. The new policy changed the output port of half of the rules (192.168.0.2-7) from port Phy1 to port Phy2. In the figures, we plot for a single experimental run on a conventional and a consistent update version of the Blueswitch design, the round-trip time (RTT) and the reception port for every data-plane packet. On the x-axes of the plots, the figure also notes the start and end times of the configuration process, which were extracted from the Blueswitch control application using the clock cycle-counter module.

It is worth noting that the default behavior of a switch on receiving a packet that does not match any rule in its match tables is to broadcast the packet on all switch ports. This behavior also occurs when an incoming packet finds the switch tables in an inconsistent state.

Figure 8b shows that the consistent scheme ensures the correct forwarding of packets during the configuration process, and that the transitions at the start and end of the configuration are smooth, with no significant latency being experienced by the measurement packets. In contrast, Figure 8a exhibits that the lack of consistency protection during an update creates transient policy violations, and that the switch resorts to the default behavior of broadcasting the packets on multiple switch ports.

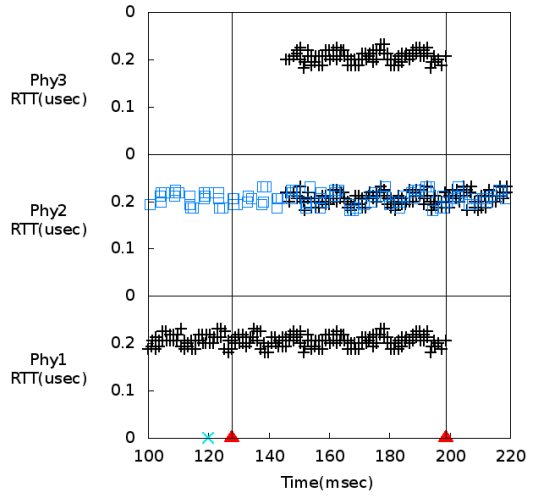
The results for the rule-insertion experiment are shown in Figures 8c and 8d. This experiment initialized the switch to load-balance traffic across three ports (Phy1, Phy2, and Phy3) of the switch using the destination IP of each packet. The reconfiguration removed eight rules that forwarded traffic to ports Phy2 and Phy3, and inserted new rules that forwarded that traffic to port Phy1, while keeping remaining traffic flows untouched. The policy configuration when performed without using a consistent scheme (Figure 8c) forwards packets according to an inconsistent policy, which mixes the old and new configuration. In contrast, the consistent scheme ensures no packets were misrouted (Figure 8d) and the data-plane RTT did not exhibit significant increase during reconfiguration.

Finally, we address the second question asked at the beginning this section, evaluating the reconfiguration latency of the Blueswitch design for variable number of rules. Figure 3 presents a comparison of the time needed to perform a reconfiguration for the two production switches and Blueswitch. The reconfiguration duration for Blueswitch is comparable with the two production switches, and is very close to that of the Arista switch. Since Blueswitch uses shadow TCAMs to store the new configuration, the forwarding performance (which uses the active TCAMs) is not impacted, and the reconfiguration process does not interfere with lookup processing. However, we point out that the configuration measurements used for Blueswitch rely on timestamps extracted from the clock module within the switch, whereas the latency measurements for the production switches include the cost of transmitting the policy changes through the control channel.

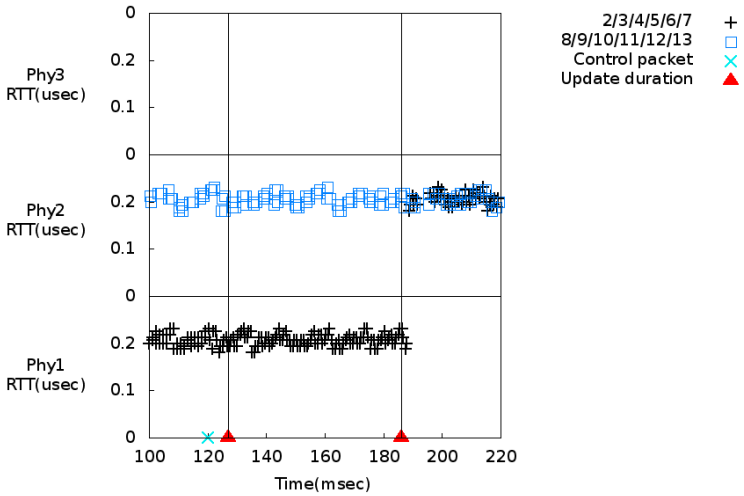
5. DISCUSSION

Datapath structure

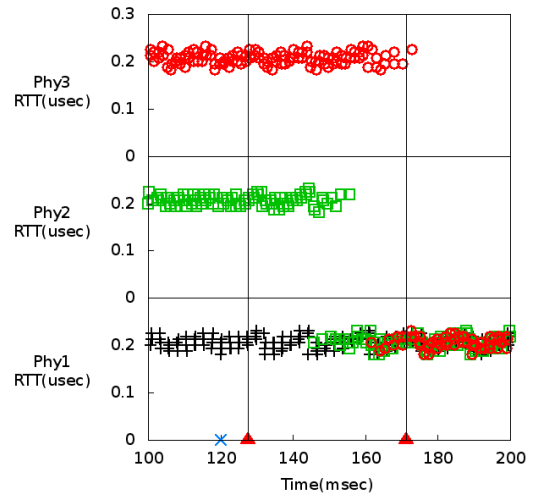
The correctness of our current Blueswitch design requires each packet header to be processed in order by each of the match tables in the pipeline. This enforces a linear sequential structure on the pipeline, which also maintains the crucial property of preserving packet order. However, the linear structure might be too restrictive for designs that wish to employ fork-join structures (as in [7]). We think our design should still be applicable as long as the join points in the pipeline have equal delays along the different paths. This condition is required to prevent later packets entering the pipeline from racing ahead of earlier ones at the joins by traversing the shorter paths, which is equivalent to packet-order preservation. We plan to prove this more rigorously and demonstrate it in a second prototype implementation.



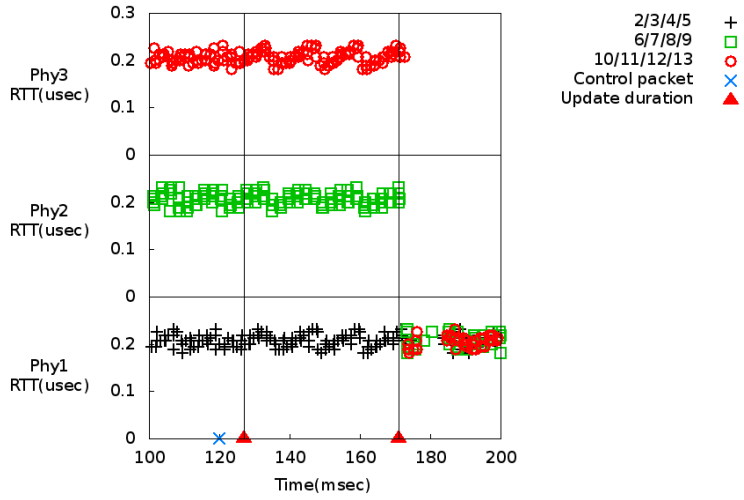
(a) Flow-modification using the packet-inconsistent interface. The unexpected traffic at Phy3 is a serious security violation.



(b) Flow-modification using the packet-consistent interface.



(c) Flow-insertions using the packet-inconsistent interface, showing both old and new rules active.



(d) Flow-insertions using the packet-consistent interface.

Figure 8: Data-plane packet behavior during policy re-configuration using the consistent and inconsistent interface. Points represent the round-trip time of a single packet as it is received through a specific port.

Area and power overhead

In ASIC implementations of switches, TCAMs are commonly the most expensive and highest power-consuming components of the chip. Our double-buffered TCAM implementation approach obviously increases such power and resource costs. However, power consumption can be reduced significantly by exploiting techniques commonly used in state-of-art ASIC implementations. Clock-gating methods can be applied to reduce the dynamic power consumption of the shadow TCAMs, which are mostly idle between configuration updates. In addition, dynamic voltage scaling (DVS) methods can reduce the static power dissipation by lowering the supply voltage without losing data in the TCAM. The effective use of these methods need to account for the frequency and size of expected configuration update transactions.

Shadow table initialization

The choice of a double-buffered TCAM structure avoids the long latencies involved when reconfiguring any active TCAMs, during which the datapath would need to be blocked. However, a double-buffered TCAM structure necessitates some mechanism of synchronizing the table entries from a newly active TCAM to the new shadow TCAM, before the shadow is reconfigured with the next configuration. Current TCAM hardware provides no efficient hardware support for this kind of table state transfer; indeed, our work in this paper indicates the value of exploring the support for this feature for future TCAM designs. In our implementation, the transfer is done by the device-driver component of the control software stack.

This state transfer can be performed immediately after a configuration commit and the swapping of the active and shadow TCAM roles. In normal operation, switch reconfigurations do not occur at a frequency comparable to the transfer delay; hence, for most configuration updates, the transfer delay has no impact on configuration latency.

Although not a motivation for our design choice, the double-buffered structure does provide hardware support for a configuration rollback mechanism.

Control communication

The major advantage of the configuration scheme is that large configuration changes can be applied in a single switch-wide transaction in a manner that is atomic with respect to the packet processing. This removes the need to factor a top-level configuration change into a series of individual entry-wise policy-safe changes separated by *Barrier* commands, reducing the number of round-trips between the switch and the controller. In the case of multi-table pipelines, the conversion of top-level single-entry modifications using algorithms like OTN [16] would require packet-consistent updates to multiple tables, which our scheme supports.

Correctness

In future work, we plan to formalize the informal correctness proof of packet-consistent configuration presented in Section 3.2 and investigate techniques for automatically linking it to the source code for the actual hardware implementation. In the context of the consistency problem in distributed network configuration updates, our scheme provides guarantees at the level of individual switch datapaths that can be used to help reason about the safety of a distributed solution.

6. RELATED WORK

As a result of the new programming opportunities afforded by the SDN paradigm, an interesting body of work has arisen to provide provably correct network management: e.g., special programming languages [3], provably correct controllers [5] and new network primitives [1]. However, work to date has assumed (often implicitly) consistent behavior at the level of individual switches.

Earlier work in configuration-consistent updates (e.g., [1], [2], [6]) treat distributed consistency from the viewpoint of multiple switches involved in a centrally administered network domain. As noted earlier (§1), these schemes use an existing field in the Ethernet frame (typically the VLAN tag or an MPLS label) to embed a configuration version and perform updates using a distributed two-phase commit using a centralized controller. As a result, they require the physical modification of every packet entering and leaving the administrative domain, but they do not specify how this is to be accomplished; indeed, a network domain may already use VLAN tags and/or MPLS labels. In addition, the two-phase commit requires that each switch table store the rule sets of both the previous and the new configurations, thus implicitly implying the same hardware resource overhead as a double-buffered scheme. Our scheme explicitly implements double buffering and stores version numbers internally in the switch. Thus, it does not need to modify packets in the network.

In summary, [1], [2], [6] choose to work with existing switch datapaths without any hardware changes, at the cost of requiring modifications to all traffic entering and leaving the network. Our approach chooses instead to modify the datapath architecture of individual switches, to gain the advantage of not needing to modify any packet in the network.

The overhead in maintaining two versions of rule configurations simultaneously in a switch can be reduced by incrementally transitioning between two configurations through a sequence of intermediate states [6], where each transition requires less rule-space overhead. A similar problem was addressed by [17] in the context of consistent updates of routing tables from BGP announcements. They considered the construction of an incremental sequence of updates that provided the same routing results as the atomic application of a batch update; they show that a general solution is not possible, and instead propose a near-optimal heuristic instead (where optimality is defined in terms of routing table size). The incremental sequence of intermediate consistent updates can also be computed dynamically [18] using a heuristic scheme; this speeds up the median configuration update speed by adapting to the different time-scales at which switches are able to perform their individual updates. All of these schemes are orthogonal to how each individual update is performed, and can be easily adapted to networks using our switch architecture.

In a distributed context, [19] address the issue of consistent composition of policies arriving from a distributed control plane consisting of fault-prone controllers. They propose a transactional interface to solve the problem of conflicting policy updates, and prove that an atomic read-modify-write operation is *necessary* to update a network consistently in the presence of controller crashes. The Blueswitch design can support precisely such an atomic update scheme.

The scheme leveraged in this paper is also inspired by other approaches to support atomicity of operations in hardware systems. To support precise exception handling [20], modern pipelined processors must effectively introduce some notion of versioning to undo speculative instructions. Hardware transactional memories [21] employ additional logic to extend versioning to span more than a few micro-operations. Our approach is far more lightweight than those approaches as packets have no enforced ordering, unlike instructions in an instruction stream, and as such we may always

commit to completing an action when we start it.

Recent innovative hardware switch architectures have tackled improved resilience (*e.g.,* [22]) and flexibility in the datapath (*e.g.,* programmable hardware parsers along with the use of multiple configurable match tables [23]). Although not the focus of that work, the consistency of the configuration interface as exposed to the datapath is unaddressed in these architectures. This is particularly crucial in designs of the latter nature, where a minor change in a top-level configuration translates into multiple match tables and parsing TCAMs updates.

7. CONCLUSION

This paper demonstrates some policy violations that can occur in a network when switches lack a configuration interface enforcing consistent configuration. To address such violations, we have presented the Blueswitch design, which provides an improved interface that is packet-consistent with respect to a modern switch datapath. This results in an implementation of the OpenFlow *Bundle* feature that supports atomic configuration updates. We have provided an informal proof of correctness for this design, and shown a concrete implementation in a modern OpenFlow switch design implemented on the NetFPGA platform. The design shows datapath and configuration performance comparable to commercial switch designs.

This approach relies crucially on invariants observed by both sides of the hardware-software interface. The invariants ensure the proper coordination of the updates from the multiple shadow buffers in the switch. As described in §3.2, to ensure that control-plane updates are consistent with respect to the datapath, hardware double-buffering techniques require a configuration protocol when multiple shadow buffers are involved, and this protocol requires that the software driver maintain its invariants. These invariants have been precisely specified, which should help considerably in the construction of reliable software control drivers.

Switches that support such consistent configuration provide a better foundation for recent work in formal network configuration and programming models, enabling reasoning about configuration correctness on complex switch datapaths.

Acknowledgements

Additional data related to this publication is available at the http://www.cl.cam.ac.uk/research/srg/netfpga/blueswitch/ data repository.

8. REFERENCES

[1] REITBLATT, M., FOSTER, N., REXFORD, J., AND WALKER, D. Consistent Updates for Software-Defined Networks: Change You Can Believe In! In *HotNets '11* (2011), ACM.

[2] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for Network Update. In *SIGCOMM* (2012), ACM.

[3] FOSTER, N., FREEDMAN, M. J., GUHA, A., HARRISON, R., KATTA, N. P., MONSANTO, C., REICH, J., REITBLATT, M., REXFORD, J., SCHLESINGER, C., STORY, A., AND WALKER, D. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013).

[4] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *POPL* (2014), ACM.

[5] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-Verified Network Controllers. In *PLDI* (2013), ACM.

[6] KATTA, N. P., REXFORD, J., AND WALKER, D. Incremental consistent updates. In *HotSDN* (2013), ACM.

[7] BROADCOM. OpenFlow - Data Plane Abstraction Networking Software. http://bit.ly/W7ZXo8, March 2014.

[8] GILADI, R. *Network Processors: Architecture, Programming, and Implementation*. Elsevier, 2008.

[9] STRINGFIELD, N., WHITE, R., AND MCKEE, S. *Cisco Express Forwarding*. Cisco Press, 2013.

[10] ARISTA. Arista 7050s datasheet. http://goo.gl/tVBcc2.

[11] PICA8. P-3922 Datasheet. http://goo.gl/INZ9cl.

[12] NetFPGA-10G. https://github.com/NetFPGA/NetFPGA-10G-live.

[13] ANTICHI, G., SHAHBAZ, M., GENG, Y., ZILBERMAN, N., COVINGTON, A., BRUYERE, M., FEAMSTER, N., MCKEOWN, N., FELDERMAN, B., BLOTT, M., MOORE, A. W., AND OWEZARSKI, P. Osnt: Open source network tester. *IEEE Network Magazine* (2014).

[14] OpenVSwitch for Blueswitch. https://github.com/pmundkur/ovs.

[15] ROTSOS, C., ANTICHI, G., BRUYERE, M., OWEZARSKI, P., AND MOORE, A. W. OFLOPS-Turbo: Testing the Next-Generation OpenFlow Switch. In *ICC* (2015), IEEE.

[16] PAN, H., GUAN, H., LIU, J., DING, W., LIN, C., AND XIE, G. The FlowAdapter: Enable Flexible Multi-Table Processing on Legacy Hardware. In *HotSDN* (2013), ACM.

[17] BANERJEE-MISHRA, T., AND SAHNI, S. Consistent Updates for Packet Classifiers. *IEEE Transactions on Computers 61*, 9 (2012), 1284–1295.

[18] JIN, X., LIU, H. H., GANDHI, R., KANDULA, S., MAHAJAN, R., REXFORD, J., WATTENHOFER, R., AND ZHANG, M. Dynamic scheduling of network updates. In *SIGCOMM* (2014), ACM.

[19] CANINI, M., KUZNETSOV, P., LEVIN, D., AND SCHMID, S. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM* (2015).

[20] SMITH, J. E., AND PLESZKUN, A. R. Implementing precise interrupts in pipelined processors. *IEEE Trans. Computers 37*, 5 (1988).

[21] MCDONALD, A., CHUNG, J., CARLSTROM, B. D., CAO MINH, C., CHAFI, H., KOZYRAKIS, C., AND OLUKOTUN, K. Architectural semantics for practical transactional memory. In *ISCA* (2006), IEEE.

[22] STEPHENS, B., COX, A. L., AND RIXNER, S. Plinko: building provably resilient forwarding tables. In *HotNets-XII* (2013), ACM.

[23] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM* (2013), ACM.