The Case for Making Tight Control Plane Latency Guarantees in SDN Switches

Huan Chen*
UESTC and Duke University

Theophilus Benson
Duke University

ABSTRACT

SDN controllers demand tight performance guarantees over the control plane actions performed by SDN switches. For example, traffic engineering techniques that frequently reconfigure the network require guarantees on the speed of gathering data from the network and the speed of reconfiguring the network. Yet, modern switches provide no guarantees for these control plane actions, e.g., inserting rules or gathering statistics. In fact, initial experiments demonstrate that unpredictability in control plane actions, specifically rule insertion, can inflate application completion times by a factor of 4X!

In this paper, we present Mercury, a framework that offers a novel method for efficiently and practically managing switch TCAM to enable strict performance guarantees. Specifically, Mercury builds on the fundamental properties of TCAMs and provides guarantees by trading-off a nominal amount of TCAM space for assured performance. Our preliminary evaluations show that with less than 10% overheads, Mercury provides guarantees of 10ms insertion time and improves application performance by a factor 2X to 5X.

Keywords

Software-defined Networking; Network Update

CCS Concepts

•**Networks** → Programming interfaces; Network management; Bridges and switches;

1. INTRODUCTION

Software Defined Networking (SDN) offers flexibility and programmatic control over the network. However, this programmatic control requires frequent modifications to the network's flow tables (TCAM). For example, traffic engineering SDN Apps, e.g. Google's B4 [20], require frequent network reconfigurations to improve network performance. Similarly, service chaining SDN Apps [10, 17] require fast reconfiguration to ensure network correctness.

Unfortunately, current SDN switches leverage traditional software and hardware components that are designed to support legacy protocols — e.g., BGP, which do not require frequent modifications to the switch's TCAM. Modern SDN switches reuse these existing hardware components, for example TCAM, which are not designed to support frequent network reconfiguration. As a result of this mismatch, running modern SDN Apps on these SDN switches can significantly degrade the performance of networked applications. Our experiments in Section 3, demonstrate that due to inefficiencies in the switches, the TCAM installation time can hurt application performance by 4X!

Existing approaches [23, 26, 30] seek to minimize TCAM insertion times by reordering rules or introducing new hardware algorithms. These approaches provide a best-effort attempt to minimize TCAM insertion latency; however, there are no performance guarantees. These approaches attack the symptoms (insertion latency) and not the root-cause (TCAM behavior), they mitigate and not eliminate the issues — large variations and unpredictability in switch performance still exist.

Unfortunately, without concrete performance guarantees for control plane actions, modern SDNs are unable to effectively support the growing number of novel use cases — critical infrastructures, cellular infrastructures, security systems, and virtual networks. For example, in 4G and 5G networks, there is a need to instantiate VoLTE connections within a predefined amount of time. Similarly, for cyber physical systems [11] there is a need for networks that make strong performance guarantees. Only by redesigning switch software and algorithms to explicitly support frequent control plane actions (with performance guarantees) can SDN support emerging SDN Apps.

In this paper, we present Mercury, a framework, for providing strict performance guarantees for control plane actions by intelligently partitioning and managing TCAM. Mercury builds on the observations that control plane actions are expensive when a flow table contains a large number of en-

*Work performed while at Duke University.

SOSR '17, April 03 - 04, 2017, Santa Clara, CA, USA

 ISBN 978-1-4503-4947-5/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3050220.3050237

tries [23, 18, 22] .

To this end, Mercury eliminates large tables by intelligently carving a monolithic TCAM into two tables: the first table is small in size and kept largely empty. This small table, called the *scratch table*, is used to serve all insertion and modification requests, thus from the perspective of these requests the TCAM is small and mostly empty. The second, a full sized table called the *regular table*. When the scratch table grows in size, Mercury proactively migrates entries from the scratch table to the regular table. By controlling the size of the scratch table, Mercury is able to make a broad range of performance guarantees.

In this paper, we take the first step towards realizing Mercury by systematically exploring the potential benefits of Mercury and investigating the challenges that arise in the design of Mercury. Our contributions are:

- Empirical Study: A systematic study of the impact of control plane actions on big data workloads across a variety of network switches.
- Design of Mercury: A strawman approach for providing strict performance guarantees over the latency of control plane actions by intelligently managing the switch's TCAM.
- Preliminary Evaluation: A preliminary evaluation of the benefits and overheads of Mercury.

2. BACKGROUND

In this section, we provide an overview of modern SDN switch design and highlight steps taken by SDN switches to perform control plane actions, e.g., rule installation, (§ 2.1), and summarize existing measurement studies on switch performance and control plane actions (§ 2.2).

2.1 Serving Control Plane Actions

Many SDN switches, specifically whitebox switches, run a Linux based OS with specialized device drivers to handle the switch's ASIC and specialized applications for various networking protocols. For example, Cumulus runs Debian with Quagga and specialized software to deal with resilient hashing. Unfortunately, many of these software and hardware components are not optimized for SDN-specific use-cases and are often highly inefficient. This inefficiency and sub-optimality impacts the latency of control plane actions.

In Figure 1, we illustrate the typical control flow for two of the three types of control plane actions defined in the OpenFlow spec [6]: switch-initiated (e.g., Packet-In) and controller-initiated (e.g., flow entry insertion – FlowMod). Here we focus on the workflow within the switches and highlight the resources that impact the control plane action's performance [1].

Switch-Generated Async Action: The ASIC can generate and send packets to the controller for a variety of reasons

[1] Most of the control logic is performed on the remote controller such as calculation of path, traffic engineering, but the local switches still have their own control procedure for the new rule installation which is described as follows.

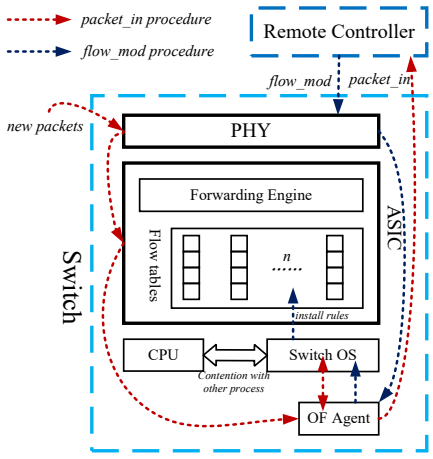


Figure 1: Operations on New Packets Arrival

ranging from the OpenFlow rule's action (e.g., forward to the controller) to a lack of matching rules. Regardless, the ASIC will raise an OS interrupt and the OS will transfer the packet from the ASIC to main memory and alert the OpenFlow agent. The OpenFlow agent will, in turn, encapsulate the packet into a *packet_in* message and send this message to the SDN controller. The performance of these actions is impacted by contention with other processes for the switch's CPU (for raising the interrupt and running the OF-agent) and memory (for copying packets).

Controller-Generated Async Action: When the switch receives control plane commands from the controller, e.g. *flow_mod* or *get_statistics*, the switch OS transfers the packets to the OpenFlow agent. For a subset of these actions, *flow_mod*, which add/modify/delete rules in TCAM, the OpenFlow agent uses the ASIC's device driver to modify rules in the TCAM's flow tables. Within a switch, the latency of performing control plane action, e.g., *flow_mod*, consists of the OS processing time (e.g., OF Agent processing, context switching) and the TCAM processing time (device driver processing and rearrangement of rules in TCAM).

2.2 Measuring Control Plane Action Performance

Recent studies [19, 23, 22] of SDN switches and TCAM performance have analyzed the performance profiles of current SDN switches. Here we summarize their key findings and use them to help motivate the design choices for Mercury.

Insertion Time Grows Linearly with number of rules In their experiments, prior work observed that rule installation time is impacted by a number of factors:

First, the priorities of the rules impacts TCAM performance [19, 23]; rules with priorities are 5-times slower than rules without priority. Furthermore, the order of insertion is important. For example, installing rules in ascending order of priorities is 10-times faster than in descending order.

Second, the number of rules in the flow tables impacts the flow insertion time [22]. For example, a flow table containing 50 rules is almost 10 times faster than the same flow table containing 200 rules. In Table 1, we present the rule

ASIC	Table Occupancy	Update/s	Performance Guarantee	Scratch Size
108 KB Firebolt-3	50	1266	3ms	50
	200	114		
	1000	23	10ms	200
	2000	12		

Table 1: Rule Update Rate of Pica8 P-3290 [22]

installation time for different flowtable occupancy levels.

Deletion is constant time operations Unlike flow insertions, deleting flows exhibits relatively trivial performance characteristics because deletions remove entries and do not require moving entries around [2]. Specifically, rule deletion latency is independent of the flow table occupancy [22] and rule priority [23].

Modifications, surprisingly, can be constant Modifications require changing the match or action of a rule – regardless they are cheap and fast because they do not require moving TCAM entries. For example, "modifying 5000 entries could be six times faster than adding new flows" claims [23]. Alternatively, modifications that alter the priority of a rule may require moving TCAM entries and perform similarly to insertions.

Takeaways The insertion time is directly proportional to the number of rules already in the flow table – we can bound the insertion time by limiting the number of rules in the table. Furthermore, there is a clear correlation between the flow table size and the max insertion time (Table 1). Finally, while there are many types of control plane actions only a few of them need to be revisited to provide strong performance guarantees. For example, flow table insertion, deletion, and modification are all part of the same control plane action, $flow\_mod$, yet we only need to explicitly design for insertions.

3. MOTIVATION

In this section, we expand on our observations and analyze the impact of control plane action latency (and variation) on networked applications (big data applications). First, we discuss our experimental setup (Section 3.1), and then we analyze the implications of control plane action latencies on big data jobs (Section 3.2).

3.1 Simulation and Models

We evaluate the impact of control plane actions using an existing flow-level event-driven network simulator [14] for simulating big data jobs, e.g., Map-Reduce. This simulator simulates the workflow of map-reduce jobs, including fetching from storage nodes (by mappers), shuffle from mappers to reducers, and writing to storage (by reducers).

Switch Performance Model. To analyze the impact of control plane actions, we introduce the control plane action latency by modifying the simulator to accurately simulate an SDN switch based on existing empirical models. Specifically, for each switch on the path of a flow, the simulator uses the distributions from related work [19, 28] to determine the insertion latencies. The insertion latency for any single rule is a function of multiple things; (1) the occupancy of the flow table, i.e., current number of rules in the flow table, and (2) the properties of the rule being inserted.

In our simulation, we model two types of switches, Pica8 P-3290 and Dell PowerConnect 8132F, using the empirically derived performance models [22]. These switch models allow us to model TCAM performance, both, control plane actions (rule installation/deletion/modification) and data plane forwarding (packet matching and forwarding latencies).

SDN Application. In our simulation, we explored two SDN Apps for improving the performance of big data workloads. The first, a proactive traffic engineering application [16] that periodically reconfigures the network by moving congested flows into other available links. The control plane action latencies are incurred when flows are migrated and the impact of this is extended periods of congestion. The second, a reactive application, installs optimal paths, based on Sinbad [13], in response to packet-in events. Control plane action latencies are incurred on flow startup and the impact is an inflation of a flow's completion time. This reactive application provides us with a worst case scenario.

Traces & Topology We run our simulator on a large scale 24 hour map-reduce trace from a 600-machine Facebook cluster [14, 13]. We use a clos-style data center topology – specifically Fat-Tree [9] with K=6.

3.2 Implications

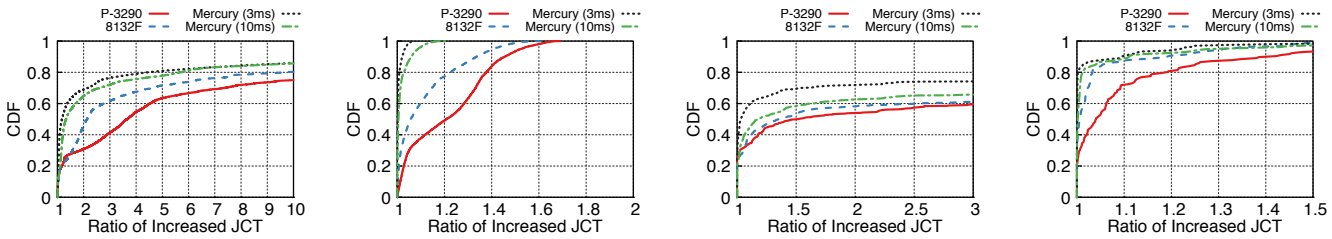
In Figure 2, we present the CDF of the increased ratio of job completion times in both proactive and reactive mode. We separate the short (Figure 2 (a)) from the long jobs (Figure 2 (b)) with reactive mode enforced and define short jobs as jobs that last less than 60 seconds and all other jobs are long jobs. Similar separation is used in the proactive mode (shown in Figure 2 (c) and (d). From the figure, we observe that short jobs are significantly impacted compared with the long jobs; the short jobs are more impacted because their flows are much shorter and are thus unable to ameliorate the latency of the control plane actions. This is particularly alarming as the short jobs are more latency sensitive and have more important flows [14].

These results further highlight the need for systematic support of performance guarantees on control plane actions. Next, we present our system, Mercury, for enforcing and maintaining these guarantees without modifying hardware or requiring forklift upgrades of the network.

4. ARCHITECTURE

To enforce performance guarantees for control plane actions, we propose a framework, Mercury, that builds on the observations discussed in Section 2. At a high level, Mercury *guarantees and bounds TCAM insertion time by restricting the size of the flow table*. To do this, Mercury maps a logical TCAM flow table into two physical flow tables: the first table, a small table (The scratch table), that's kept relatively empty and the second table (the regular table), a large table. The size of the scratch table is a function of the required performance guarantee; lower latency guaran-

[2]Deletion may result in gaps in the TCAM



(a) Short flows (b) Long flows (c) Short flows (d) Long flows

Figure 2: Job Completion Times for Reactive SDN App (a & b) and Proactive SDN App (c & d).

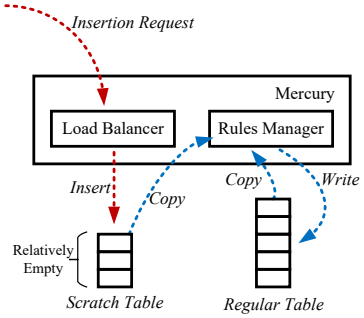


Figure 3: Mercury Architecture (Single Table)

tees require smaller scratch tables. For example, according to Table 1, a 3ms guarantee requires a scratch table size of 50 entries where as 10ms requires 200 entries.

Mercury uses two components, *Load-Balancer* and *Rule-Manager*, to manage both physical tables and provide the abstraction of one logical table. In Figure 3, we present the architecture of Mercury. Mercury intercepts all control plan actions to the TCAM: insertion, modification and deletion. In managing both tables Mercury's components performs two high level tasks:

First, the Load-Balancer handles insertions (*flow_mod*) and inserts the rules into the scratch table when the scratch table is not full. When the scratch table is full, the Load-Balancer places flows into the regular table (red arrows in Figure 3).

Second, the Rule-Manager tries to keep the scratch empty by periodically migrating rules from the scratch table to regular table (blue arrows in Figure 3). The goal of the Rule-Manager is to ensure that rules are migrated from the scratch table before the scratch table becomes full. [3]

4.1 Load-Balancer

The Load-Balancer manages insertion across both tables and ensures that both tables emulate a single logical table. To achieve these goals, Load-Balancer tackles the following two challenges:

Correctness Guarantees: Mercury guarantees that the combined behavior of the scratch and regular is identical to that of a logical table. This guarantee is complicated by the fact that new rules are generally always into the scratch table and lookup occurs first to the scratch table. Thus a newer lower priority rule may be used instead of an older higher priority rule because the newer rule is in the scratch and the older rule is in the regular table. For example, in Figure 4, two rules inserted into two tables yields a different actions than when inserted into a single table.

We intend to tackle this challenge by modifying the overlapping rules to ensure that they are inserted in a manner that respects priorities while preserving correctness. To this end, we plan to explore an efficient data structure that detects overlapping rules and selectively rewrites these rules to eliminate overlaps.

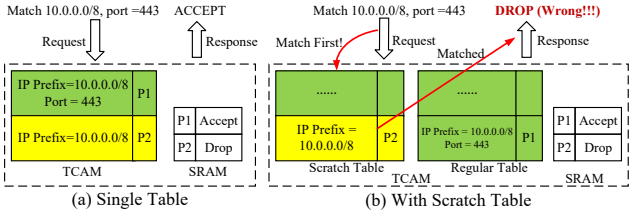


Figure 4: Correctness Challenge: Rule overlap.

Performance Guarantees: Mercury is unable to provide performance guarantees when a batch of rules larger than the scratch size are inserted instantaneously into the switch. Inserting such a large number rules will force Mercury to fill up the scratch and start inserting into the regular table. To prevent this scenario, we plan to extend the Load-Balancer to perform admission control (thus rate limiting insertion) and develop an expressive API that enables Mercury to inform applications of the rate limits associated with the requested performance guarantees.

4.2 Rule-Manager

The Rule-Manager periodically migrates the rules from the scratch table to the regular table. The goal of the Rule-Manager is to ensure that rules are migrated from the scratch table before the occupancy of the scratch table exceeds the target size. The design of the Rule-Manager must tackle several challenges: (1) Decide when to migrate rules from scratch table? (2) How should the rules be migrated to maintain consistency? (3) How does Mercury ensure that guarantees are maintained during rule migration?

Next, we elaborate on these challenges:

4.2.1 When to migrate rules?

To provide performance guarantees Mercury must migrate rules from the scratch table to the regular table before the number of rules in the scratch table exceeds its target size. This requires an intelligent technique for capturing and pre-

[3]Exceeding the target size will lead to performance violations

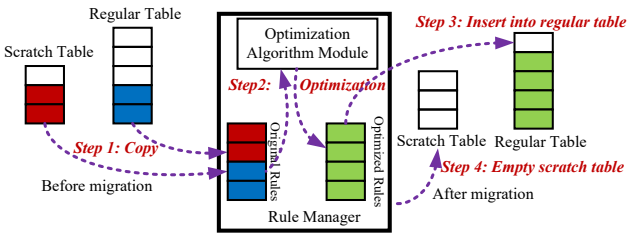


Figure 5: Rule-Manager's Migration Workflow

dicting the flow insertion rates. This technique allows Mercury to predict when the number of rules in the scratch table will exceed its limit and subsequently trigger a migration.

There are three alternatives: first by exploring predictive algorithms, e.g., estimated weighted moving average (EWMA), autoregressive model (ARM), Cubic Spline, second by modifying the controller to provide hints as to its expected insertion rate [23], and third by specifying a static threshold.

In our design of Mercury, we plan to explore the first approach for its simplicity and elegance: this approach leaves the controller abstractions unmodified and frees the programmer from the burden of determining the insertion rate.

4.2.2 How to migrate rules?

The Rule-Manager performs migration in four steps (Figure 5). First, rules from the scratch table and the target flow table(s) are copied into the Rule-Manager. Then, the Rule-Manager optimizes the rules using existing algorithms [30]. Third, Rule-Manager copies optimized rules into the regular flow tables. Finally, the scratch table is emptied. During the entire migration process, the scratch table remains usable.

The migration process is designed to ensure that the Rule-Manager maintains the following properties:

Performance Guarantees: Provided the scratch table is emptied and a rule is never inserted when it is full, Mercury will be able to provide performance guarantees.

A challenge arises when rules are inserted faster than Mercury can empty out the scratch table. For example, if a batch of rules are inserted such that the batch size is larger than the scratch table, then Mercury will be unable to provide guarantees for a subset of the rules. To address this issue, we plan to explore the use of a rate limiter. Similar to existing QoS primitives, Mercury will provide specific performance guarantees as long as the application's insertion rate is below a predefined limit. As part of future work, we intend to develop principled methods for determining the optimal rate-limit for the different performance guarantees.

Correctness During Migration Consistency: To ensure correctness, Rule-Manager does not empty the scratch (step 4) until after migration (step 3). This ensures that at any point in time there is at least one rule to process and service packets. However, there may be two identical rules in both tables: one in the scratch table and one in the regular table. Fortunately, the default behavior for Mercury is to stop matching after a packet matches an entry in the scratch table.

4.3 Implementation Feasibility

While we have not implemented Mercury in hardware, we have discussed our design with Broadcom engineers and switch vendors. Our discussion indicates that Mercury can be implemented in the current line of switches using interfaces readily available in the Broadcom SDK. [4]

Modern and traditional SDN switches [6, 5, 2] provide control over partitioning of TCAM tables, called TCAM carving. For example, Cisco [8, 4] subdivides TCAM into 8 predefined slices and provides operators with commands for repartitioning (or resizing) these slices. More modern switches provide richer control and flexibility. The Broadcom SDK allows operators to "carve" the TCAM: specifically, determine the number of entries in each slice and the size of the keys for each slice. These slices are subsequently mapped to groups and assigned priorities. Lookup into a TCAM table is done in parallel across all slices with each slice returning at most one match [1, 4]. The TCAM resolves conflicts across different slices using the configured priorities.

To support Mercury, we can carve the TCAM into two slices. Both slices are configured with identical keys; however, the scratch slice (scratch table) is configured to be significantly smaller than the regular slice (regular table). The TCAM is configured to resolve conflicts in favor of the scratch slice. During lookups, both slices are analyzed in parallel, this parallel lookup results in the correct answer. During insertions, Mercury tries, first, to insert into the group/slice allocated for the scratch table and if this fails because the scratch table is full, then it attempts to insert the rule into the group/slice allocated for the regular table.

4.4 Discussion

For deletion and modifications, Mercury acts slightly differently. For deletion, Mercury deletes the actions associated with the rule – this ensures that deletion takes no time and the empty rules can be removed during the optimization that happens in the migration phrase. For modifications, Mercury applies them without concerns because they inconsequential incur great latencies.

Multiple TCAM Tables: While Mercury is designed to provide the abstraction of a single logical flow table, modern switches and OpenFlow specifications support multiple flow tables. Mercury can be extended to support multiple physical flow tables by providing multiple logical flow tables. Each logical table is mapped to a scratch and a regular slice – both slices are embedded in a physical table. This independent decomposition of the different tables enables Mercury to provide different guarantees for the different table: a feature that may be particularly attractive when the different tables are used for radically different functionality [29, 24]. To preserve the semantics of the original pipeline, Mercury configures each regular table to exhibit the default "miss" behavior of the original table – either go to the next table, send the packet to the controller, or drop the packet.

[4]SDK access is required because existing interfaces to switches, e.g. OFDPA [3] and OpenNSL [7], do not provide control over configuration of TCAM slices.

5. PRELIMINARY ANALYSIS

Next, we revisit the simulation presented in Section 3 and explore the benefits and overheads of employing Mercury.

Application Level Benefits. In Figure 2, we present the performance improvements provided by employing Mercury configured to provide 3ms and 10ms insertion latency guarantees. Note, regardless of the switches used, Mercury provides strict performance bounds. We observe that, in reactive mode, Mercury with 10ms latency guarantee significantly improves the median and tail by 2X-5X. Moreover, Mercury with 3ms latencies provides improvements over Mercury with 10ms. We observe that, with the proactive SDN App, the benefits provided by Mercury are not as pronounced as with the reactive SDN App. The benefits are less pronounced with the proactive SDN App because there are fewer network modifications and thus Mercury's framework is used less frequently.

Flow Level Benefits. Next, to better understand Mercury's performance under the proactive SDN App, in Figure 6, we examine microscopic properties. Specifically, focusing on flow completion times (FCT) and flow installation times (FIT). We observe that the flow installation times (Figure 6 (a)) are largely constant with minor variation. These minor variations exist because while Mercury provides an upper bound on performance, Mercury may, in fact, perform better than this upper bound and thus installation time may be lower than expected.

In Figure 6 (b), with 10ms guarantee we can observe that the median flow completion time is reduced by 48% and 80% over the 8132F and the P-3290 switches respectively. This echoes observations made at the application level.

Overheads. The overhead of employing Mercury is directly proportional to the configured performance guarantees. According to existing studies [22] (summarized in Table 1), the Pica8 P-3290 switch incurs a 2.5% space overhead and the Dell 8132F switch incurs a 6.67% space overhead to provide a 3ms insertion delay guarantee — for 10ms, both switches incur a smaller overhead. Mercury trades a modest amount of TCAM resources to provide the application and flow level gains described earlier.

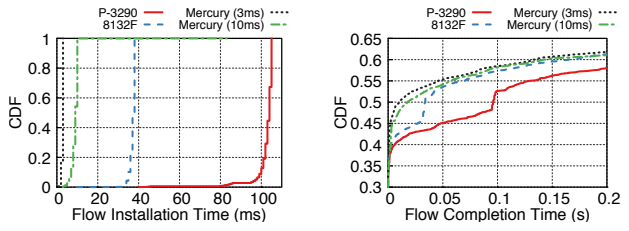


Figure 6: Performance in Proactive Mode

6. RELATED WORK

Modeling Switch Performance: Prior works [18, 23, 27, 15] have conducted empirical studies on the factors that impact control plane action latency. These studies motivate our work. We build on these studies by demonstrating the impact of these control plane actions on network applications.

Maximizing TCAM Performance: To improve TCAM performance, existing approaches either re-order rules [23, 25, 26, 21] or change the TCAM insertion algorithms [30]. Even though these approaches reduce control plane action latency, they do not provide any guarantees or assurances. Mercury address exactly this: Mercury provides performance guarantees over control plane action latencies.

7. DISCUSSION AND FUTURE WORK

Next, we discuss ongoing work to extend Mercury to provide performance guarantees for other types of SDN control plane actions and programmable data planes.

GetStatistics/PacketOut/PacketIn The performance of these messages is largely a function of contention for the switch's CPU and memory resources. To provide guarantees, we intend to optimize the switch's software stack by eliminating unnecessary bloat and introducing mechanisms that allow the switch to reserve and allocate resources for these control plane actions.

Emerging Programmable data planes Current prototypes and designs are based on properties of modern merchant silicon and ASICs. Yet, we believe that the core design of Mercury is applicable to the emerging generation of programmable data planes, e.g., P4 and RMT chips [12], because these platforms reuse traditional TCAM-based flow tables and Mercury addresses a property of TCAM that is invariant to underlying TCAM design.

8. CONCLUSION

SDN Apps, e.g. traffic engineering and service chaining, require frequent modification to the TCAM using control plane actions. Moreover, many of these SDN Apps require their actions to be completed in a timely manner. Unfortunately, modern SDN switches do not provide concrete performance guarantees for such control plane actions – instead, the switches provide a best-effort service. To support many emerging applications and scenarios, we must redesign switch software and algorithms to explicitly support frequent control plane actions.

In this paper, we propose Mercury, a system which provides strict performance guarantees for control plane actions by intelligently partitioning and managing TCAM flow tables. Mercury provides these guarantees by trading off a modest amount of TCAM space for enhancing TCAM performance. Our preliminary evaluations show that with less than 10% overheads, Mercury is able to provide significant improvements (2X to 5X improvements).

9. ACKNOWLEDGMENTS

We thank Collin Dixon, our shepherd, Brent Stephens, and the anonymous reviewers for their invaluable comments. This work is partially supported by NSF grant CNS-1409426, National Basic Research Program of China 2013CB329103, NSFC fund 61271165.

10. REFERENCES

[1] Acl solutions guide. http://extrcdn.extremenetworks.com/wp-content/uploads/2014/10/ACL_Solutions_Guide.pdf.

[2] As5712. http://www.edge-core.com/productsInfo.php?cls=1&cls2=8&cls3=44&id=15.

[3] Broadcom sdn solutions of-dpa. https://www.ietf.org/proceedings/90/slides/slides-90-sdnrg-3.pdf.

[4] Nexus 9000 tcam carving. http://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html.

[5] Noviswitch. http://noviflow.com/products/noviswitch/.

[6] Openflow switch specification v1.3. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[7] Opennsl. https://github.com/Broadcom-Switch/OpenNSL.

[8] Understanding and configuring switching database manager on catalyst 3750 series switches. http://www.cisco.com/c/en/us/support/docs/switches/catalyst-3750-series-switches/44921-swdatabase-3750ss-44921.html.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, 2008.

[10] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *Proc. ACM SOSR*, 2015.

[11] R. Baheti and H. Gill. Cyber-physical systems. *The impact of control technology*, 12:161–166, 2011.

[12] 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. In *Proc. ACM SIGCOMM*, 2013.

[13] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. ACM SIGCOMM*, 2013.

[14] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *Proc. ACM SIGCOMM*, 2014.

[15] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM*, 2011.

[16] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. Transparent and flexible network management for big data processing in the cloud. In *Proc. USENIX HotCloud*, 2013.

[17] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proc. ACM SIGCOMM HotSDN*, 2013.

[18] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan. Mazu: Taming latency in software defined networks. *University of Wisconsin-Madison Technical Report*, 2014.

[19] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring control plane latency in sdn-enabled switches. In *Proc. ACM SOSR*, 2015.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proc. ACM SIGCOMM*, 2013.

[21] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[22] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about sdn flow tables. In *Proc. PAM*, 2015.

[23] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *Proc. ACM CoNEXT*, 2014.

[24] Y. Li, G. Yao, and J. Bi. Flowinsight: Decoupling visibility from operability in sdn data plane. In *Proc. ACM SIGCOMM*, 2014.

[25] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zupdate: Updating data center networks with zero loss. In *Proc. ACM SIGCOMM*, 2013.

[26] P. Perešíni, M. Kuzniar, N. Vasić, M. Canini, and D. Kostiū. Of.cpp: Consistent packet processing for openflow. In *Proc. ACM SIGCOMM HotSDN*, 2013.

[27] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *Proc. PAM*, 2012.

[28] D. Tai, H. Dai, T. Zhang, and B. Liu. On data plane latency and pseudo-tcp congestion in software-defined networking. In *Proc. ANCS*, 2016.

[29] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen. Umon: Flexible and fine grained traffic monitoring in open vswitch. In *Proc. ACM CoNEXT*, 2015.

[30] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu. Ruletris: Minimizing rule update latency for tcam-based sdn switches. In *Proc. IEEE ICDCS*, 2016.