ESPRES: Transparent SDN Update Scheduling

Peter Perešíni†, Maciej Kuźniar†, Marco Canini‡, Dejan Kostić•*
†EPFL ‡Université catholique de Louvain •KTH Royal Institute of Technology
†<name.surname>@epfl.ch ‡marco.canini@uclouvain.be •dmk@kth.se

ABSTRACT

Network forwarding state undergoes frequent changes, in batches of forwarding rule modifications at multiple switches. Installing or modifying a large number of rules is time consuming given the performance limits of current programmable switches, which are also due to economical factors in addition to technological ones.

In this paper, we observe that a large network-state update typically consists of a set of sub-updates that are independent of one another w.r.t. the traffic they affect, and hence sub-updates can be installed in parallel, in any order. Leveraging this observation, we treat update installation as a scheduling problem and design ESPRES, a runtime mechanism that rate-limits and reorders updates to fully utilize processing capacities of switches without overloading them. Our early results show that compared to using no scheduler, our schemes yield 2.17-3.88 times quicker sub-update completion time for 20th percentile of sub-updates and 1.27-1.57 times quicker for 50th percentile.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software-Defined Networking; Performance; Update scheduling

1. INTRODUCTION

Changes to network forwarding state are frequent—new network users must be accommodated, network policies change, routing is engineered to adapt to traffic conditions, virtual machines (VMs) are constantly spun up, torn down or moved in the cloud, maintenance events and failures cause topology changes, etc. To ensure that updates complete with minimum disruption, recent works [6, 11, 13] focused on the data plane consistency guarantees that can be provided as a network transitions from one state to another.

However, very little consideration has been given so far to another important problem: *optimizing the installation of forwarding rules to allow a vast fraction of flows to be processed according to the updated network state as soon as possible*[1]. As Software Defined Networking (SDN) substantially enhances network programmability, we expect that the rate of network updates will increase even further. For example, tight flow packing to achieve maximum link utilization in Google's B4 [5] can require frequent changes. On the other hand, installing or modifying a large number of rules across a pool of (potentially heterogeneous) switches can be a time-consuming operation due to the substantial latencies incurred in processing rule operations on the switch and updating switch chips accordingly. These latencies are due to hard-to-overcome technological as well as economical factors [2].

In this paper, we propose to compensate these inefficiencies by tackling update installation as the problem of scheduling which operations are going to be sent to which switch at any given moment. Solving this difficult scheduling problem is important because updates are often on the critical path, *e.g.*, for implementing policy changes or service provisioning. We present the initial design and preliminary evaluation of ESPRES, a runtime mechanism that carefully plans the installation of individual updates and actively manages switch message queues, while striving to fully utilize message processing capacities of switches without overloading them. In doing so, ESPRES leverages the observation that a large update typically consists of a set of independent sub-updates, and hence sub-updates can be installed in parallel, in any order.

ESPRES introduces a *per-switch virtual message queue* that is realized as the combination of the message queue on the switch extended with a message queue maintained at the controller. Our key insight is that this queue extension enables ESPRES to continuously reassess at the controller the order in which messages should be sent to switches, since once the messages are queued at the switches they can no longer be reordered with the current versions of OpenFlow or similar protocols. ESPRES' queue manager observes how long each switch takes to execute rule operations, and carefully issues enough of them to keep the switch occupied without excessively queuing messages there.

*Work done when the author was with IMDEA Networks.

HotSDN'14, August 22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620747 .

[1]Our initial motivation for the problem appeared in [12].

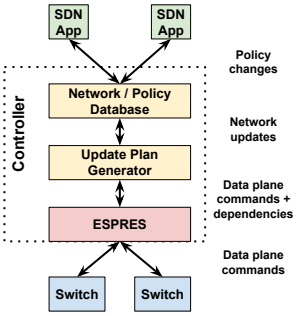


Figure 1: Position of ESPRES in an SDN architecture.

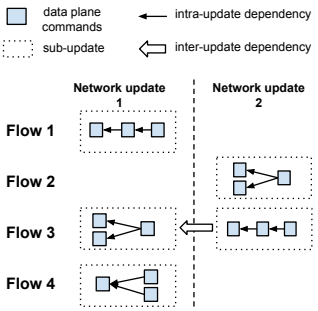


Figure 2: A key observation: *network updates* can be broken down into independent sub-updates.

At a high-level, ESPRES first groups all operations of an update into *sub-updates* that affect different logical traffic flows, and then schedules operations by ordering the *sub-updates*. ESPRES aims to execute as many *sub-updates* in parallel as possible while offering the ability to optimize for different goals. For example, the goal of finishing flows sooner is accomplished by choosing shorter updates first. On the other hand, ESPRES can reduce rule overhead in the switches by preferring *sub-updates* that remove rules first.

Throughout this paper, we assume an SDN controller architecture consisting of two layers: the *Network / Policy Database* and the *Update Plan Generator*. To a good extent, this architecture is a simplification from existing SDN control plane proposals such as Onix [8] and ONOS [1]. SDN applications interact with the controller through a northbound interface that allows them to programmatically alter the network state via modifications to the Network / Policy Database. Changes to the database are propagated to the *Update Plan Generator*, a component that translates these changes into actual commands affecting switch rules (hereafter *rule operations*) as well as a *dependency graph* (Fig. 2) that describes the intra-update dependencies between operations [11]. Additionally, inter-update dependencies between different *network updates* may exist. In a typical SDN stack, an OpenFlow driver or similar component would then send rule operations to switches based on the computed update plan. As shown in Fig. 1, ESPRES operates below the *Update Plan Generator* and subsumes this latter component. ESPRES receives a stream of *network updates* and optimizes the efficiency of rule installation by scheduling *rule operations*. Finally, ESPRES acknowledges all finished *network updates* back to the *Update Plan Generator*.

Our early results show that compared to a no-scheduler baseline, a simple ESPRES scheduler yields 2.17-3.88 times quicker *sub-update* completion time for the 20th percentile of *sub-updates* and 1.27-1.57 times quicker for 50th percentile. Moreover, an ESPRES scheduling algorithm optimizing for rule-space overhead causes only 3.5-17% overhead instead of 62% with no scheduler.

2. SCHEDULING WITH ESPRES

Our key observation is that an SDN update is typically induced by one or more high-level events, such as *e.g.*, traffic engineering (TE) recomputations, VM migrations, and topology or policy changes. Typically, these events result in a batch-style update of forwarding state spanning multiple switches. Importantly, a network update easily consists of a set of sub-updates that are independent of one another, that is, there are no rule installation dependencies between rule operations corresponding to these sub-updates. For example, each TE-affected flow can typically be shifted independently of any other flow.[2]

Independence between sub-updates plays an important role because any combination of independent sub-updates can be applied in an arbitrary order, or even interleaved in parallel, without introducing data plane inconsistencies (*e.g.*, causing a forwarding loop or a blackhole, imposing mutually exclusive forwarding actions, or violating other safety conditions [11]). Such independence is a great source of flexibility for choosing an order in which rule operations and whole sub-updates are applied. By leveraging such flexibility, ESPRES can optimize the *network update* installation for a variety of goals.

However, to leverage all this flexibility in practice, ESPRES faces the challenge of handling switches with different (and variable) performance characteristics. This requires ESPRES to be able to quickly compute the ordering of rule operations and adapt it based on current conditions. This is a major departure from previous schemes that work on coarser granularity (*e.g.*, split an update into several rounds but do not schedule within a round) in which the complete update installation is precomputed [6, 10]. As a result, our design consists of two main components: (*i*) the *Queue Manager*, which is responsible for keeping switch "service times" short, and (*ii*) the *Scheduler*, which is responsible for choosing the order in which rule operations are performed.

2.1 Managing switch command queues

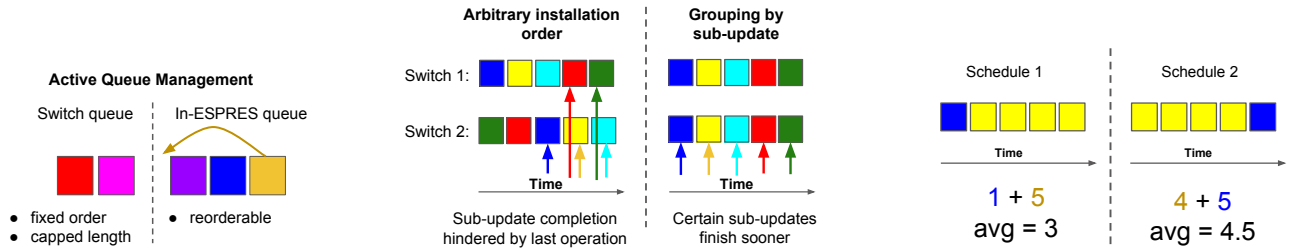
A key to exploit all the available scheduling flexibility is maintaining good switch responsiveness by actively managing their command queues. That is, instead of sending all commands at once to a switch (and queuing them there with no possibility for future reordering or cancellation), ESPRES queues these commands at the controller and sends to the switch only a small subset of them (Fig. 3a). Naïvely sending all available commands to a switch fills up its queue, which delays installation of some rule dependencies. Instead, when the queue length is actively managed, ESPRES can decide which commands are to be sent next according to a particular scheduling discipline (*e.g.*, prefer rules from sub-updates that already started).

Because sending rule operations to a switch is not an instantaneous process, the *Queue Manager* needs to trade-off queue length (queuing adds additional delay as well as limits reordering possibilities) versus switch performance.

In particular, a very short queue length ensures low waiting latency but causes low rule modification throughput, whereas a very long queue provides full throughput at the expense of rule operations being stuck in the back of the queue for a long time. In our prototype, we use a simple heuristic for queue management: our algorithm limits the number of outstanding requests for each switch not to exceed a fixed threshold (5 in our experiments).[3] We validate

[2]Even with congestion free-induced limitations, there are solutions generating sets of independent sub-updates [10].

[3] Because OpenFlow lacks positive acknowledgments, we limit outstanding requests by using barriers and tracking



(a) Active queue management helps with queue re-ordering.

(b) Grouping rules by sub-update helps finishing part of a *network update* sooner.

(c) Ordering sub-updates by size can speed up average sub-update installation time.

Figure 3: Overview of techniques ESPRES uses to schedule rule operations.

our decision for using just a simple threshold as well as the trade-off between latency and performance in Section 3.

2.2 Scheduling rule operations

When the *Queue Manager* considers a particular switch command queue to be short, it notifies the *Scheduler*. Then, the *Scheduler* selects which rule operation should be next sent to that switch. A baseline solution is to disregard the active queue management and scheduling altogether, and send rule operations as soon as all their dependencies are met. Such baseline solution has however a major drawback. Observer that each traffic flow starts following the new desired forwarding configuration only after all rules corresponding to the particular network flow are installed, *i.e.*, the sub-update for that flow completes. Therefore, if rule operations are ordered in an arbitrary (random) way, sub-update completion will be frequently hindered by the last rule operation. Instead, grouping rule operations by sub-update helps finishing parts of the *network update* sooner as illustrated in Fig. 3b.

2.2.1 ESPRES schedulers

We base our schedulers on the observation that it is beneficial to install all rule operations of a given sub-update at roughly the same time. Thus, we design ESPRES schedulers as sub-update schedulers — they decide on the *preferred* order in which the sub-updates should be installed. Note that this is not a strict sequential ordering — ESPRES can send rule operations from any (even the last) sub-update as long as previous sub-updates do not have rule operations which are *ready* to be sent on a per-switch basis. A rule operation is ready if all of its dependencies are already installed.

Preferred order scheduler. The basic version of the scheduler always sends the first ready rule operation, according to the preferred order (we discuss how this order is derived in the next section). That is, each time the *Queue Manager* informs the *Preferred order scheduler* that (some) switches are ready, the scheduler goes through the sub-updates in the current order, inspects each sub-update and sends out any ready rule operations. The scheduler ends iterating through sub-updates when the end of the sub-update list is reached or when there is no more switch queue space, whichever comes first.

Batch-ready scheduler. We observe that we can further improve the *Preferred order scheduler* performance by using the following heuristic. Instead of sending any ready rule operation across sub-updates as soon as a switch is available, we can synchronize ready operations within a sub-update and send them at the same time. The *Batch-ready scheduler* thus iterates through the sub-update list similarly to *Preferred order scheduler* but sends sub-update ready rule operations as a batch and only when all corresponding switches are available. This provides an effect similar to gang scheduling [3], an operating systems scheduling concept that enhances the performance of multi-threaded programs by co-scheduling multiple threads of the same program at the same time.

2.2.2 Ordering sub-updates

The preferred ordering in which sub-updates should be installed plays an important role in the scheduling performance. For example, if the goal is to finish updating the majority of sub-updates in the network as soon as possible, we should order shorter sub-updates first similarly to the shortest job first scheduling in the context of operating systems (see Fig. 3c).

ESPRES supports a variety of preferred orderings. A baseline is an arbitrary fixed order of sub-updates (*e.g.*, sort on sub-update identifier). ESPRES can also sort on the sub-update size, priority (if given by the controller), or a custom-defined order. Moreover, the preferred order is not necessarily fixed throughout the whole update.

For instance, to support the goal of minimizing mid-update rule overhead,[4] ESPRES uses an estimate of the current switch rule overhead to prefer updates that remove rules from the currently most overloaded switches. In this case, ESPRES periodically reorders all sub-updates according to a penalty function after potentially installing a sub-update. We define the penalty function as:

$$\sum_{s \in switches} \left(\max(current\_rules_s - target_s,\ 0)\right)^2$$

where $current\_rules_s$ is the current number of rules installed at switch s, and $target_s$ is the maximum of rules at s before the entire update starts (known by the controller) and the number of rules after it ends, *i.e.*, $target_s = \max(initial_s,\ initial_s + update\_delta_s)$

the number of sent `BarrierRequest` messages and received `BarrierReply` messages.

[4]Controllers using the two-phase consistent update [13] require keeping both old and new rules at the same time.

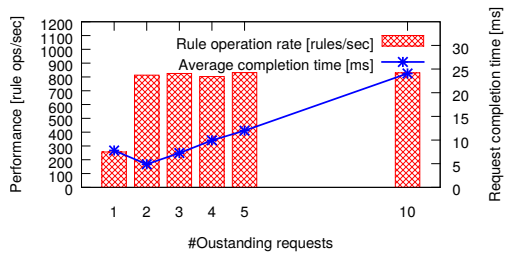


Figure 4: Performance of a limit-outstanding-requests queue management on a Pica8 switch. Rule modification throughput is constant for queue longer than 1.

flows	10 rules/s	100 rules/s	1000 rules/s
100	2.17x / 1.27x	2.17x / 1.27x	2.17x / 1.27x
1000	3.53x / 1.49x	3.53x / 1.49x	3.53x / 1.49x
5000	3.88x / 1.57x	3.88x / 1.57x	3.88x / 1.57x

Table 1: Sub-update finish time improvement (at 20th and 50th percentile) with ESPRES compared to no scheduler on IBM topology. ESPRES helps for all update sizes, but the improvement increases with the size. Switch speed has no effect on the relative improvement.

3. EVALUATION

We implemented our ESPRES prototype as a Python program on top of the POX OpenFlow controller platform. We also developed a discrete event simulator to evaluate the system. First, to validate the simulator, we run ESPRES in an emulated Mininet environment using the reference OpenFlow switches rate limited to 40 rule modifications/second, which corresponds to existing hardware switches [4]. As the results in simulation and emulation are comparable, we use the simulator in our experiments to fully control all parameters. Further, unless specified otherwise, we set the rule modification rate at switches to 1000 rules/second, which is much more than current generation of OpenFlow switches is capable of. Although fast switches are adversarial to ESPRES, we choose to use such value to confirm that ESPRES will be equally relevant in future deployments.

3.1 Active queue management

First, we explore how short the switch queue can be without decreasing control plane performance. We design a benchmark where the controller maintains a fixed number of outstanding requests per switch and measures the switch performance. The controller pre-populates a switch flow table with 500 initial rules and then repeatedly removes a random rule and replaces it with a new one followed by a barrier request.

We run the benchmark on a Pica8 3290 switch (PicOS 2.0.4, OVS 1.10.0) and summarize the results in Fig. 4. The main takeaway is that using a small number of outstanding requests (*e.g.*, two) does not decrease control plane performance. Running the benchmark on an HP ProCurve E5406zl, we observe that it requires a few more outstanding requests, and we thus set the number of outstanding requests in our experiments to 5 (to account for some variance in switch performance).

3.2 Intra-update scheduling

To show the scheduling benefits of ESPRES, we evaluate it in three scenarios with different scheduling goals.

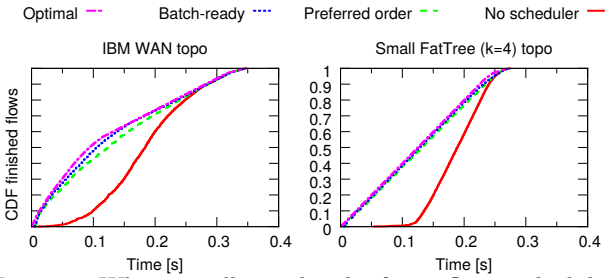


Figure 5: When installing a batch of new flows, scheduling reduces completion time for most of them.

Improving mean time to finish. We first focus on the mean time to finish a sub-update. This is an important metric when the controller is installing a batch of new flows (*e.g.*, spinning up a new VM) or repairing existing flows after a topology change. In this experiment, ESPRES coordinates installation of new sub-updates, each representing a flow on a shortest path between two edge switches chosen at random. Flows are installed in a per-packet consistent manner [13], *i.e.*, we use a two-phase update where the ingress rule is installed only after all other rules are in place. We run the experiment using various parameters, as discussed below. Each run was repeated 3 times and we observed comparable results across runs; therefore, due to space limit, we highlight a few major results.

Fig. 5 shows the CDF of flow installation time for 1000 flows in an IBM topology [7] with 18 switches (all switches are edge switches) and a FatTree topology with 20 switches ($k = 4$, 8 ToR switches are edge switches). Table 1 summarizes results of experiments across a range of flows and switch performance parameters.

Overall, the results show that significant benefits come even from our simple scheduling algorithms. The Batch-ready scheduler comes very close to an optimal schedule calculated by an integer linear program,[5] which has a high run-time overhead of 10 minutes. Further, our algorithm is 3.5 times better than not using a scheduler[6] for 20th percentile of flows, 1.5 times better for 50th percentile and achieves equal total update time. Note that scheduling does not introduce performance benefits in the 9x-percentiles of flow installations, which correspond to the flows that are installed last. This is because these flows ultimately depend on which bottleneck switch is last to finish with rule installations. Nonetheless, scheduling does not worsen the installation time of these flows. Moreover, as shown in Table 1, increasing the switch rule installation speed does not affect the scheduling benefits. The overall update time decreases proportionally, but the benefits coming from ESPRES remain. Finally, the benefits of ESPRES is greater for bigger updates when the scheduler has higher freedom to reorder sub-updates.

Lowering mid-update switch rule overhead. In these experiments, we assess rule overhead using a version of the

[5]We assume a priori known and constant switch performance.

[6] "No scheduler" is a scheduler that iterates over flows in some predefined order and sends all ready operations to the switch as soon as possible, ignoring any queue management. We try with different flow orderings but the ordering itself seems to have no major effect on the results.

Scheduler \ Sub-updates	352		1074	
per-switch overhead	max	avg	max	avg
No scheduler	55.4%	26.3%	62.3%	27.8%
Inc. consist. updates [6]	15.8%	15.0%	17.1%	8.7%
ESPRES	16.8%	5.7%	3.5%	1.3%

Table 2: Comparison of rule overhead when using two-phase update methodology [13] to achieve consistency.

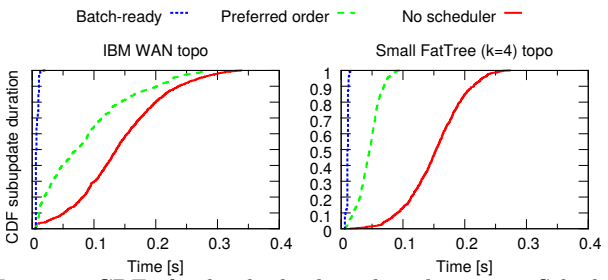


Figure 6: CDF of individual sub-update durations. Scheduling helps minimizing the possible disruption times.

scheduler that tries to minimize the mid-update switch rule overhead.[7]

We design an experiment that resembles a TE recomputation in a FatTree topology with 20 switches. We first setup a number of flows over arbitrary shortest paths between pairs of ToR switches chosen at random. We consider both 500 and 1500 flows. Then, we compute new paths for all flows by randomly choosing a new shortest paths for each flow. In about 25% of cases the same path for a flow is chosen and therefore the number of sub-updates is lower than the number of flows (*e.g.*, 352 and 1074). Finally, ESPRES installs all new flows in a consistent manner, *i.e.*, we provide dependencies between rule operations of the same flow as a three stage update where we (*i*) add new rules, (*ii*) then modify the "ingress" switch, *i.e.*, the switch where the new path diverges from the old one, (*iii*) delete old rules.

Table 2 summarizes our results. We observe that a naïve update without a scheduler results in major maximum per-switch overhead (up to around 60%), while both incremental consistent updates [6] configured to use 4 rounds and ESPRES keep the overhead low (17% in the worst case). Further, ESPRES outperforms incremental consistent updates for larger updates because ESPRES reacts to current switch conditions at run-time and interleaves rule installations and rule deletions from different sub-updates to lower the overhead. We note that ESPRES is a best-effort service and does not guarantee low worst-case overhead like incremental consistent updates does. In the future, we plan to explore the benefits of running ESPRES on top of incremental consistent updates to bound the worst-case behavior while maintaining the benefits of run-time information.

Minimizing sub-updates durations. We conclude the evaluation of scheduling goals with an experiment measuring the duration of individual sub-update installation times, *i.e.*, the time taken to install a sub-update from start to finish, as measured at the controller. We use the same setup as for the mean time experiment. Results in Fig. 6 suggest that our scheduler rapidly decreases the in-progress sub-update durations for most sub-updates, while finishing the update at the same time as the baseline case with no scheduling. This reduction can be attributed to grouping operations per sub-update and co-scheduling all operations at the same time.

[7]We calculate the per-switch overhead as in [6]: overhead in percent $= 100 * (worst/base - 1)$ where *worst* is the maximum number of rules during the update and *base* = max(pre-update rule count, post-update rule count).

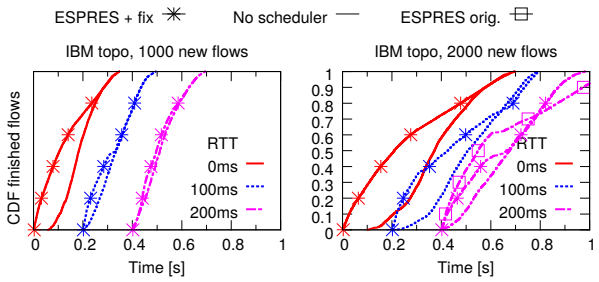


Figure 7: CDF of flow installation time as we vary the controller-switch RTT.

3.3 ESPRES sensitivity to network latency

To check how our greedy scheduler performs when working with delayed information, we vary the controller-switch round-trip-time (RTT) from 0 to 200ms. To avoid switch underutilization, we adjust the number of outstanding requests to cover at least RTT-worth of rule operations (*e.g.*, 2, 105 and 205 outstanding requests for 0ms, 100ms and 200ms, respectively.)

Understandably, the update time increases with higher RTTs, as shown in Fig. 7. However, we also observe that ESPRES performs worse than no scheduler for long RTTs (*e.g.*, the line marked with squares in the figure). An investigation revealed that our schedulers are too eager to finish the already-started sub-updates, which does not interact well with the long latencies during the final phase of an update as some sub-updates do not start until the very end and then they have to wait long for their dependencies. In particular, for each operation denote its *depth* as the maximum length of dependency chain from the operation to the update end (depth is 1,2 or 3 in our experiments).

Assuming the same RTT between controller and all switches, if an update contains an operation at depth d, the update cannot finish sooner than in $d \times RTT$. To fix the scheduler, for each switch s and each depth d, we calculate $pending(s,d)$ as the count of all operations at depth d still not sent. Then we calculate

$$T_{min}(s,d) = d \times RTT + pending(s,d)/rate(s)$$

and

$$T_{ETA} = \max_s \left(\sum_d pending(s,d)/rate(s) \right)$$

If $T_{min}(s,d) \times (1+\varepsilon) > T_{ETA}$ for some depth d, there is a risk that an update would be delayed because of operations with depth d (we use $\varepsilon = 5\%$ as a safety margin). In this case, we force the scheduler to send to switch s only operations with depth $\geq d$, effectively starting new sub-updates instead of finishing already started ones.

After fixing the scheduler to start sub-updates earlier if some operations may wait too long because of dependencies, the results improve. There are two main conclusions coming from Fig. 7. First, the fixed scheduler performs much better than baseline early during an update and stays no worse than the baseline near the end. Second, the time when the

scheduler changes the strategy depends on the RTT, the switch rule modification rate and the update size. Thus, even if switches become faster, ESPRES will still be helpful in the future because the updates are likely to grow and the RTTs to decrease.

4. RELATED WORK

Reitblatt *et al.* [13] introduce the notion of consistent network updates, and propose a two-phase update approach. Katta *et al.* [6] reduce consistent updates' rule overhead in switches by operating in a set of rounds [6], albeit increasing the overall update duration. zUpdate [10] performs congestion-free updates using a set of carefully computed steps. ESPRES improves upon these works by lowering the time at which the majority of sub-updates are installed. Mahajan and Wattenhofer [11] recently discuss consistent updates in SDN, and their *plan executor* subsystem is perhaps the closest in spirit to our work. However, ESPRES goes further in offering the initial design, implementation, and evaluation of a network-wide scheduler for rule installation.

Huang *et al.* [4] measure switch performance (*e.g.*, rule installation rate) to build high-fidelity switch models. Jive [9] measures the performance of OpenFlow switches according to predetermined patterns to derive switch capabilities; these capabilities could in turn be used to optimize network behavior. In contrast, ESPRES dynamically adapts to runtime switch performance while scheduling rule installation. ONOS [1] aims for quick rule installation, and can potentially leverage ESPRES.

5. SUMMARY AND FUTURE WORK

We presented ESPRES, a transparent layer for scheduling switch commands such as forwarding rules that arise in large network updates. ESPRES is the first system that adapts to switch command plane performance at runtime. By doing so it enables a vast majority of the flows to begin functioning correctly much quicker when compared to launching all commands on the switches at once as is typically the case today.

As part of future work, we are working on extending ESPRES to accommodate several other use-cases.

Inter-update scheduling. A strawman solution to install multiple updates is to serialize them by the time of arrival. As different updates may have bottlenecks on different switches, we argue that it is beneficial to install independent sub-updates from different updates in parallel. However, while providing dependencies within a single update is a responsibility of the *Update Plan Generator*, we cannot expect it to provide us with dependencies between all already submitted but not fully applied updates. Instead, ESPRES can infer inter-update dependencies if the *Update Plan Generator* annotates each sub-update with a *slice definition*, *i.e.*, a description of which switches and which parts of the header space the sub-update is touching. Then, a non-empty intersection indicates a potential dependency relationship between sub-updates.

Once the inter-update dependencies are identified, our schedulers could keep a *ready set* of sub-updates and schedule rule operations only from this set.[8] The scheduler moves a sub-update to the ready set as soon as all sub-updates it depends on are installed. This allows the scheduler to use the same scheduling techniques and assume sub-update independence as it is the case within a single update. Finally, to avoid starvation, we prefer older updates by including update epoch number as a primary sorting key for our schedulers.

Prioritization. Not all network updates are equally important. For example, failure recovery should be prioritized over traffic engineering, even if the latter was sent to ESPRES first. However, scheduling such updates is an intricate problem – the scheduler needs to balance priority requirements with liveness (low priority updates should not be starved indefinitely). Furthermore, scheduler might be forced to install a low-priority update because high-priority updates depend on them.

[8]For performance reasons, the scheduler might also want to re-compute the preferred order only for this set.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110. This work was (partially) supported by the ARC grant 13/18-054 from Communauté française de Belgique.

6. REFERENCES

[1] ONOS: Open Network Operating System, 2014. `http://tools.onlab.us/onos-learn-more.html`.

[2] A. Curtis, J. Mogul, J. Tourrilhes, and P. Yalagandula. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[3] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16, 1992.

[4] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-Fidelity Switch Models for Software-Defined Network Emulation. In *HotSDN*, 2013.

[5] S. Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[6] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[7] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Journal on Selected Areas in Communications*, 29(9), 2011.

[8] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[9] A. Lazaris, D. Tahara, X. Huang, L. E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Jive: Performance Driven Abstraction and Optimization for SDN. In *ONS*, 2014.

[10] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate : Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[11] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[12] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. ESPRES: Easy Scheduling and Prioritization for SDN. In *ONS*, 2014.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.