Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization

Aggelos Lazaris[+] Daniel Tahara[×]
Xin Huang[⋆] Li Erran Li[†] Andreas Voellmy[×] Y. Richard Yang[•×] Minlan Yu[+]
[†]Bell Labs, Alcatel-Lucent [⋆]Cyan Inc. [+]USC [•]Tongji University [×]Yale University

ABSTRACT

A major benefit of software-defined networking (SDN) over traditional networking is simpler and easier control of network devices. The diversity of SDN switch implementation properties, which include both diverse switch hardware capabilities and diverse control-plane software behaviors, however, can make it difficult to understand and/or to control the switches in an SDN network. In this paper, we present Tango, a novel framework to explore the issues of understanding and optimization of SDN control, in the presence of switch diversity. The basic idea of Tango is novel, simple, and yet quite powerful. In particular, different from all previous SDN control systems, which either ignore switch diversity or depend on that switches can and will report diverse switch implementation properties, Tango introduces a novel, proactive probing engine that infers key switch capabilities and behaviors, according to a well-structured set of Tango patterns, where a Tango pattern consists of a sequence of standard OpenFlow commands and a corresponding data traffic pattern. Utilizing the inference results from Tango patterns and additional application API hints, Tango conducts automatic switch control optimization, despite diverse switch capabilities and behaviors. Evaluating Tango on both hardware switches and emulated software switches, we show that Tango can infer flow table sizes, which are key switch implementation properties, within less than 5% of actual values, despite diverse switch caching algorithms, using a probing algorithm that is asymptotically optimal in terms of probing overhead. We demonstrate cases where routing and scheduling optimizations based on Tango improves the rule installation time by up to 70% in our hardware switch testbed.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet switching networks*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Management*

Keywords

Software-defined Networking; OpenFlow; Switch Diversity

CoNEXT'14, December 2–5, 2014, Sydney, Australia.

http://dx.doi.org/10.1145/2674005.2675011 .

1. INTRODUCTION

A major benefit of software-defined networking (SDN) over traditional networking is simpler and easier control of network switches. In particular, OpenFlow [11] has provided a standard, centralized mechanism for a network controller to install forwarding rules at the forwarding engines (called flow tables) of a heterogeneous set of network switches, substantially reducing controller-switch dependency, and hence control programming complexity.

One key issue that such a standard controller-switch protocol cannot resolve, however, is the diversity of switch implementation properties, which we define to include both hardware capabilities and control software behaviors. In particular, the hardware of switches can have significant differences in their physical structures such as TCAM size, which significantly affects forwarding throughput over large sets of rules; the software implementing the control of switches can differ substantially as well, such as in their TCAM management algorithms and flow installation efficiency. Since such diversity reflects many factors, including real-life, random, idiosyncratic designs as well as systematic switch vendor engineering exploration (which can be essential to foster switch vendor innovation), it is inconceivable that all switches will have the same hardware capabilities and software behaviors.

The presence of SDN switch diversity can pose significant challenges in the control of SDN switches. In particular, we identify two challenges. First, diversity creates an understanding challenge. Although newer versions of OpenFlow [16] allow switches to report configurations and capabilities (called features), the reports can be inaccurate. For example, the maximum number of flow entries that can be inserted is approximate and depends on the matching fields (*e.g.*, IPv4 vs. IPv6). Furthermore, many important configurations and behaviors are not reported, for example, whether a switch uses a software flow table and the caching policy that determines when a rule should be in the software flow table or TCAM.

Second, diversity can create a substantial utilization challenge, which can lead to control programming complexity. For example, consider two switches with the same TCAM size, but one adds a software flow table on top. Then, insertion of the same sequence of rules may result in a rejection in one switch (TCAM full), but unexpected low throughput in the other (ended up in the software flow table). Now consider that the two switches have the same TCAM and software flow table sizes, but they introduce different cache replacement algorithms on TCAM: one uses FIFO while the other is traffic dependent. Then, insertion of the same sequence of rules may again produce different configurations of flow tables entries: which rules will be in the TCAM will be switch dependent. Whether a rule is in TCAM, however, can have a significant impact on its throughput, and therefore quality of service (QoS).

In this paper, we design Tango, a novel approach to explore the issues of SDN control in the presence of switch implementation diversity. Tango addresses both the understanding challenge and the utilization challenge.

The basic idea of Tango is novel, simple, and yet quite powerful. In particular, different from all previous SDN programming systems, which ignore switch diversity or at most simply receive reports of switch features (in newer version of OpenFlow), Tango is designed with the observation that instead of depending on switch reports, one can use real measurements to achieve better understanding of switch diversity. Based on this observation, Tango introduces a novel, proactive probing engine that measures key properties of each switch according to a well-structured set of *Tango patterns*. Each Tango pattern consists of a sequence of standard OpenFlow flow modification commands and a corresponding data traffic pattern, based on the general knowledge of the switch architecture. For example, a Tango switch TCAM capacity discovery pattern sends data packets matching the inserted rules. If Tango detects larger delay of the packets for newly installed rules, Tango has detected the TCAM size.

Utilizing the measurement results from the Tango patterns, Tango derives switch capabilities as well as the costs of a set of equivalent operations that can be utilized, through *expression rewriting*, to optimize networks with diverse capabilities and behaviors. For example, the Tango priority pattern inserts a sequence of rules in ascending priority and descending priority in two separate tests. In the hardware switch we tested, Tango records that the descending pattern has a substantially longer latency than that of the ascending pattern. The difference is about 6 fold if Tango priority pattern has 5,000 rules. The Tango scheduling algorithm automatically exploits these Tango patterns. Therefore, if Tango has 5,000 rules to install on the hardware switch, it will write the sequence in the ascending order to achieve low rule installation latency. Second, comparing across switches, Tango records that insertion into the flow table of the hardware switch is substantially slower than into that of the software switch. Hence, when Tango needs to install a low-bandwidth flow where start up latency is more important, Tango will put the flow at the software switch, instead of the hardware switch.

We use both experiments on three types of hardware switches and simulations with Open vSwitch (OVS) to evaluate Tango. Tango inference engine is highly accurate. In particular, our inference accuracy of flow table size is with 5% of actual value. For a real switch, Tango can reduce rule installation time by a factor of 12.

We emphasize that despite the progress made by Tango, its scope is still limited, focusing mainly on performance. Additional switch diversities, such as ability to provide different functionalities, remain to be explored.

The remainder of this paper is organized as follows: Section 3 motivates our work by first examining standard switch architecture and then proceeding to explore the diversity of their implementation. Section 4, 5, and 6 provide a concrete specification for the Tango system, the inference process used by the Tango controller to understand the switch diversity, and an example Tango scheduler. We provide evaluations of our inference algorithms and performance optimizations in Section 7, explore related work in Section 8, and conclude in Section 9.

2. BACKGROUND: TYPICAL SWITCH ARCHITECTURE

We start with some background on a typical architecture of switches that implement the OpenFlow specification, which is the key SDN controller-switch protocol. The OpenFlow specification only defines an interface between a controller and a switch supporting OpenFlow. It leaves switch vendors the freedom to implement the internals in their convenient ways and perform optimizations, as long as the rule matching and traffic processing semantics are preserved. As a result, switches can have diverse implementations in their software layers and on how much they leverage the hardware interfaces. Such diversity leads to varying switch behavior and performance, both on the control plane and the data plane. In this paper, we use three proprietary switches from three vendors as both examples and evaluation targets. We refer to these three vendors as Vendor #1, Vendor #2, and Vendor #3; their switches as Switch #1, Switch #2, and Switch #3 respectively.

Figure 1 illustrates the architecture of an OpenFlow enabled hardware switch. It usually includes three key components:

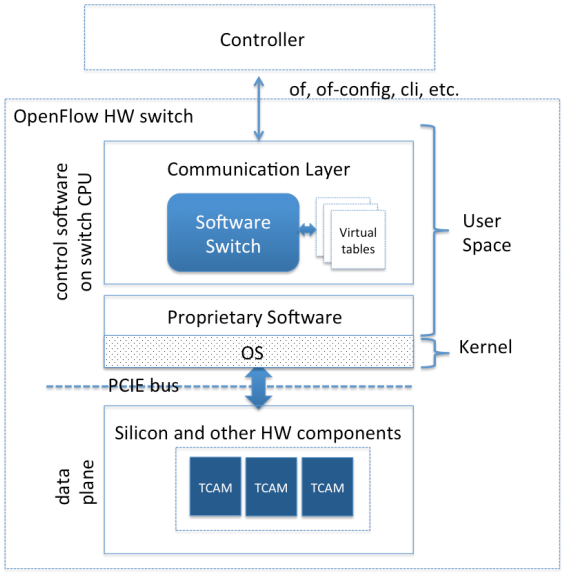


Figure 1: OpenFlow switch software stack architecture.

Communication Layer: This layer is responsible for the communication between the switch and remote controllers. It consists of software agents supporting standard network programming and management protocols such as OpenFlow, OF-CONFIG, and SNMP. Most vendors use the OVS (Open vSwitch [17]) userspace code as the OpenFlow agent, and then perform vendor-specific extensions to better match OVS to their platform design. This is where interesting things happen and we will get back to this topic in details at the end of this section.

Proprietary Layer: Below the communication layer is the proprietary layer, in which a vendor provides its own software to translate from the platform independent model (*e.g.*, OpenFlow model) to platform specific commands and configurations. This layer includes components such as line card manager, driver, and vendor SDK. The proprietary layer obtains access to the switch hardware via switch OS. Similar to a computer system, switch OS is responsible for managing switch hardware resources (*e.g.*, switching ASIC, TCAM, and Bus) as well as the communications among software layers and modules (*e.g.*, between OVS and proprietary software).

Although there are proposals on developing and open sourcing generic an OpenFlow hardware abstraction layer (HAL) across heterogeneous switching platforms [22], these projects are still at very

early stage. Today, the proprietary software is still mainly developed and maintained by individual switch vendors.

TCAM and Switch Hardware: Modern switch silicon consists of multiple physical tables (*e.g.*, L2 MAC table, L3 LPM table, and TCAM) that are shared among multiple pipeline stages along the fast path packet processing. Although potentially all these tables could be used as OpenFlow flow tables, so far most switch vendors appear to have leveraged only TCAM. Even for switches (*e.g.*, Switch 1 in this paper) that support OpenFlow 1.1 and higher, the multiple tables in OpenFlow pipelines are mostly implemented in switch software. Only entries belong to a single table are eligible to be chosen and pushed into TCAM.

As described above, vendor-specific implementations usually happen at the communication layer or the proprietary layer. For example, Switch 1 maintains multiple virtual tables in user space and uses a certain algorithm to figure out which entries should be pushed down to TCAM and when. Both tables could be used for packet forwarding. TCAM facilitates fast path packet forwarding and virtual tables are used for slow path packet forwarding. Only packets that hit neither of the tables will be sent to the controller. Switch 2 chooses to have the TCAM table as the only flow table. That is, packets that are able to match TCAM entries will be forwarded in the fast path. All other packets will be forwarded to the controller. Switch 1 can afford more rules than the number of entries that can be held in TCAM, but traffic processed in slow path will suffer from longer delay and less throughput. As a contrast, the number of entries that can be supported on Switch 2 is limited by the on-board TCAM but all matching packets will be guaranteed to go through fast path and be forwarded in line rate.

At the proprietary layer, vendors often have different switch software stack designs (*e.g.*, multi-threaded vs. single-threaded processing, synchronous vs. asynchronous message passing, memory hierarchy and management, data sharing mechanism). All these design and implementation details could potentially impact OpenFlow control channel performance (e.g., OpenFlow packet generation and processing rate) or the slow path packet-processing rate. At the same time, vendors are also left with the freedom to perform optimization for OpenFlow (*e.g.*, putting user space virtual tables into the kernel, taking advantage of DMA to remove switch CPU from critical packet processing pipeline).

All these variations contribute to the varying performance and behavior of the OpenFlow hardware switches. We will take a closer look at a few key performance and behavior differences in the next section using real experimental results.

3. SWITCH DIVERSITY: EXAMPLES

Despite the fact that many switches use the typical architecture presented in the preceding section, their details differ substantially, and such differences can have large impacts.

Diverse flow installation behaviors: For OpenFlow switches that maintain more than one table, switch vendors must implement algorithms to manage the assignment of flow entries to different flow tables, and maintain certain mapping between the tables. This is totally vendor dependent.

For example, OVS based software switch makes decision based on data plane traffic. Whenever a flow entry is pushed down to the switch, it first gets installed in the user space software table. Only when data plane traffic matches this entry, an exact match entry will be pushed to the kernel table [17]. This explains the three-tier delay in Figure 2(a). In this experiment, we first install 80 non-overlapping OpenFlow entries. Then, we start generating 160 data plane flows, with 2 packets per flow, and measure the delay. Among the 160 flows, the first 80 match the existing entries, while the rest do not. Since the entries are installed in user space table only, the first packet of any flow will not match any kernel table entry, and thus will be forwarded one level up, to the user space table (i.e., slow path). It will then match an entry in the user space table, get processed and forwarded to the right output port. At the same time, an exact match entry will be installed in kernel so that the following packets belongs to the same flow (i.e., the second packet) will be processed in the fast path. Flows that do not match any existing entries will be forwarded and processed by the controller. Thus, three levels of delay are shown in the plot: fast path and slow path for the first 80 flows, and control path for rest of the flows. OVS uses this simple 1-to-N mapping (i.e., one user space entry could map to multiple kernel space entries) between the two tables. There are more complex caching algorithms proposed [7].

Switch	user space software tables		TCAM/kernel tables	
	L2/L3	L2+L3	L2/L3	L2+L3
OVS	$< \infty$	$< \infty$	$< \infty$	$< \infty$
Switch #1	$< \infty$	$< \infty$	4K	2K
Switch #2	None	None	2560	2560
Switch #3	None	None	767	369

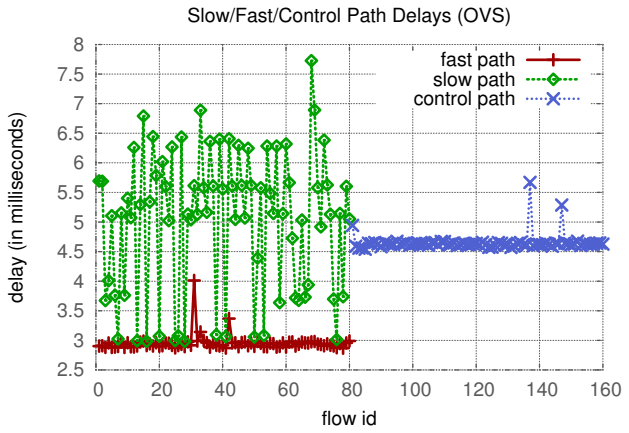
Table 1: Diversity of tables and table sizes.

In contrast, the software table in Switch #1 works as a FIFO buffer for TCAM. That is, the oldest entry in the software table will be pushed into TCAM whenever an empty slot is available. This is observed in Figure 2(b). In this experiment, we first install 3500 non-overlapping flow entries. Then we generate a total more than 5000 data plane flows (again, two packets per flow) and make the first 3500 flows match installed entries. Again, we observe three-tier delays, but there is no delay difference between the first packet and the second packet of a particular flow. This indicates that unlike the OVS case, flow entry allocation here is independent of the traffic. The first 2047 flows have much shorter delay than rest of the flows. This means the first 2047 entries (there is an default route pre-installed when switches get connected with controllers) are installed in TCAM and the corresponding flows are forwarded through fast path. The rest entries reside in user-space software table(s), and thus the corresponding flows are forwarded through slow path. When a flow does not match any of the existing flow entries, it will be forwarded to controller and get processed there. Although FIFO is simple, it may lead to policy violation problem [7] when there are overlapping entries in both tables and the higher priority ones are in software tables.

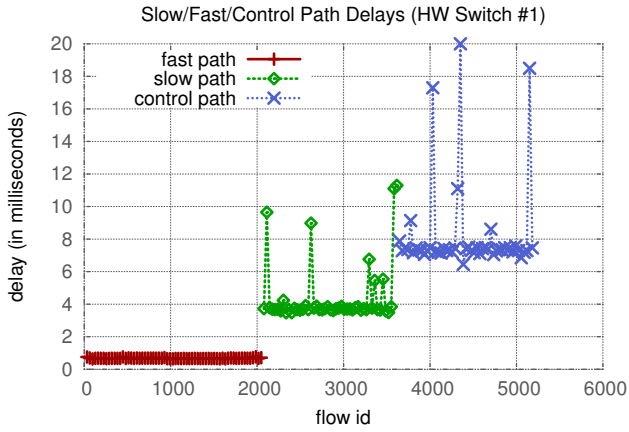
Switch #2 only has one table (TCAM). It corresponds to the two-tier delay in Figure 2(c) . Flows matching TCAM entries are forwarded in fast path, while flows matching no TCAM entries will be forwarded and processed by controller at control path.

Diverse flow tables and table sizes: Although all switches must expose flow tables as their forwarding engine model to the controller, per OpenFlow specifications, different switches may map this model to different tables. In particular, we can identify three types of tables: user space software table, kernel software table, and TCAM hardware table. Table 1 shows the types of flow tables of four OpenFlow switches, including an OVS based software switch and three HW switches from different vendors that cover a variety of switch implementations.

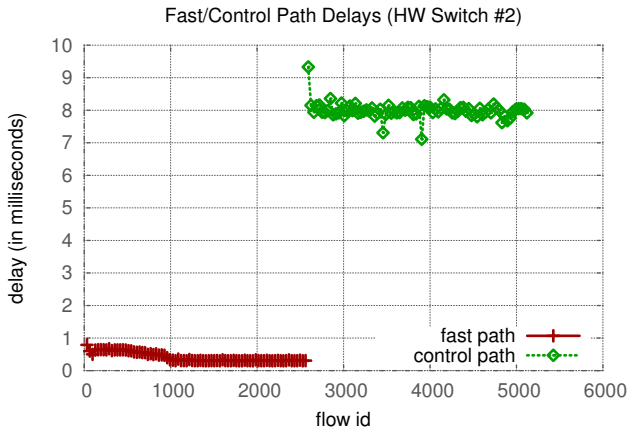
As shown in Table 1, OVS maintains software tables in both user space and kernel space. Switch #1 has both a hardware table implemented in TCAM, and can support 256 virtual tables in the user space. In contrast, switches 2 and 3 have only TCAM tables, but no software tables.



(a) Three tier delay in OVS.



(b) Three tier delay in Switch #1.



(c) Two tier delay in Switch #2.

Figure 2: Various edge detection algorithms.

Even if two switches have the same type of tables, their tables may have different number of entries. Although we could assume software tables have virtually unlimited size, this does not apply to TCAM tables. TCAMs are expensive and power hungry, thus TCAM size on a switch is quite limited and varies from one model to another. Moreover, the number of entries that can be stored in a fix-sized TCAM depends on: 1) entry type, and 2) operation mode. For example, in single-wide mode, entries can match only L2 headers or only L3 headers, while in double-wide mode, entries can match both L2 and L3 headers. When configured in the double-wide mode, TCAM can only hold half as many entries as in the single-wide mode.

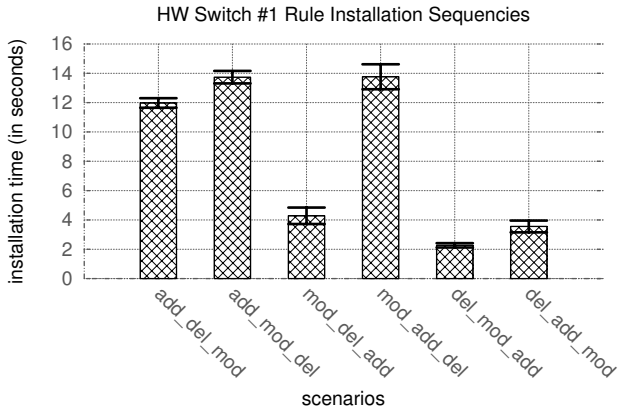
It is interesting to see how this would impact the flow table size on different switches. Switch #1 allows the user to configure the TCAM mode. If being configured in L2 only or L3 only mode, it could have 4k L2 or L3 only flows. However, if we want to match on both L2 and L3, we can only get 2K entries installed in TCAM. Switch #2 and #3 does not allow user to customize TCAM mode. However, the experiment results in Table 1 reveal that the TCAM in Switch #2 has been configured as double-wide mode, since we can only install 2560 flow entries no matter whether the entries are L2 only, L3 only, or a mix of L2 and L3. This obviously is not an efficient design under the scenario that most of the flow entries are either L2 only or L3 only. Switch #3 is a better design in this aspect, since it could automatically adjust the table size according to entry type. When all flow entries are L2 only or L3 only, it could hold 767 flow entries comparing with only 369 L2 and L3 entries.

Unlike TCAM (or other hardware tables), software tables in either user space or kernel space can support virtually unlimited number of entries, but their matching and forwarding performance will be limited by CPU capability and the type of rules we install [13].

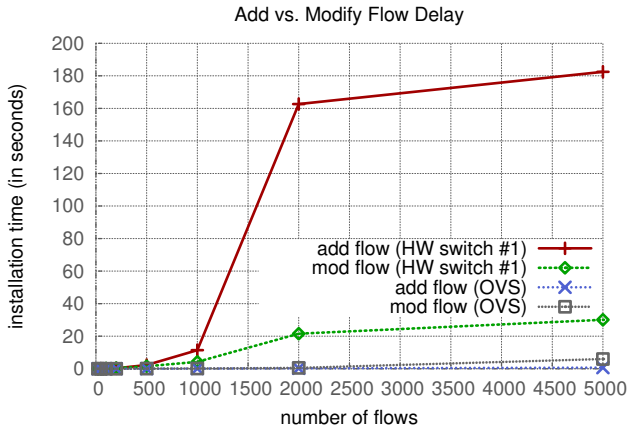
Diverse delay with different orders of rule operations: We measure the installation latency of various numbers of rules in various orders using Switch #1, Switch #2, and OVS. For Switch #1, and OVS, we preinstall 1000 rules in random priority, and for Switch #2 we preinstall 500 rules (in order not to exceed the TCAM capacity of the hardware switches). Then, we measure the speed to perform a set of flow addition, deletion and modification operations with different permutations with the same start and end sets of rules. Specifically, we implement all the six possible permutations of 200 flow additions, 200 deletions, and 200 modifications. We repeat the experiment 10 times and we present the average over the 10 experiments. The result for the hardware Switch #1 is shown in Figure 3(a).

Diverse delay using different flow tables: from above description, it is easy to understand that packets match different flow tables encounter different forwarding delays. In general, we expect fast path delay to be much smaller than slow path delay. Control path delay should be the longest since packets need to traverse between separate boxes.

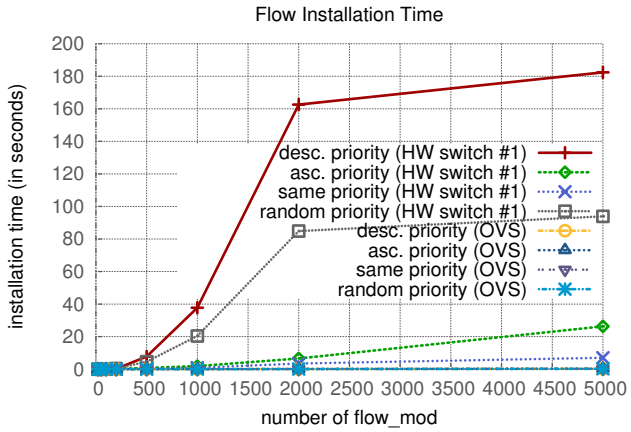
As shown in Figure 2(a), OVS based software switch has three-tier delays: a constant fast path delay of 3 ms, a varying slow path delay of 4.5 ms on average, and a control path delay of 4.65 ms. This numbers do not differ much. The varying slow path delay was resulted from the variation of switch CPU utilization to install the micro-flow at the kernel space. Similarly, Figure 2(b) has three-tier delays. The fast path delay is consistent around 0.665 ms, which is much faster than the slow path delay (typically 3.7 ms) and control path delay (7.5 ms on average). Here the fast path could process and forward packets in line rate. Figure 2(c) shows that the fast path performance and the control path performance of Switch #2



(a) Total flow installation time for the case of 200 adds, 200 mods, and 200 dels for HW Switch #1.



(b) Performance of mod vs. add for an OF HW Switch.



(c) Flow installation time for different priority patterns in OVS and a OF HW Switch. (The four curves for OVS overlap with each other.)

Figure 3: Rule installation time under different scenarios.

are comparable with Switch #1, with a data path delay of 0.4 ms on average and a control path delay of 8 ms on average.

Diverse controller-switch channel performance: The controller - switch interactions are also diverse in key aspects. First, different rule operations (e.g., add new entries or modify existing entries) have different performance. This is especially true for hardware OpenFlow switches that use TCAM in fast path forwarding. Figure 3(b) shows how long it takes to either install n new entries or update n existing entries, where n varies from 20 to 5000. In order to support the priority concept defined in OpenFlow specification, entries in TCAM need to be sorted, with higher priority entries on top. When a packet comes, it will be processed according to the first TCAM entry it hits. In this case, adding a new entry may lead to shifting multiple existing TCAM entries. This is not needed when modifying an existing entry. From Figure 3(b), we observe that modifying 5000 entries could be six times faster than adding new flows under our setup.

Second, different rule priorities have different performance, especially for hardware switches. Inserting the same number of new flows with exactly the same priority takes the least amount of time. This is because increasing priority avoids shifting existing TCAM entries to make room for new entries. Figure 2(c) shows that it takes significant less time to insert 5000 new flow entries in ascending priority order than in descending priority order. So far, we have observed 46X performance boosting by comparing the descending with constant priority ordering in the case of 2000 entries. Even by comparing random with ascending priority ordering, we can get 12X performance boosting to install 2000 non-overlapping entries. In addition, Figure 2(c) shows the results from the same experiment in OVS where we observed that priority has no impact on the rule installation delay.

The above observations have significant importance even in the cases where proactive rule installation is used (e.g. failover rules), since there are applications like traffic engineering that still requires a very fast rule update in order to adapt to the new network conditions.

4. TANGO ARCHITECTURE

Overview: The real measurement results in the preceding section provide both motivations and inspiration for the design of Tango. In other words, real measurements can give insights on the diversity of switches, and hence can form a foundation of understanding and handling SDN switch diversity. Measurement based programming is not new. For example, many programming systems first run test scripts to test the capability and availability of features of a platform, to handle platform diversity. The key novelty of Tango is that we apply this methodology in the context of handling SDN switch diversity, which poses its own unique challenges: (1) What are the measurements to reveal switch capabilities and behaviors? (2) How to hide, as much as possible, switch diversity?

Components: Figure 4 shows the basic components and information flow of Tango. The key component is the central Tango Score and Pattern Databases (TangoDB), where each Tango pattern is defined as a sequence of OpenFlow flow mod commands and a corresponding data traffic pattern. As an extensible architecture, Tango allows new Tango Patterns to be continuously added to the database.

Given a Tango Pattern and a switch, the Probing Engine applies the pattern to the switch, and collects measurement results. The collection of switch measurements can be either offline testing of the switch before it is plugged in the network, but online testing when the switch is running. The measurement results are stored into a central Tango Score Database, to allow sharing of results across components. All components in Tango can send requests to the Probing Engine to apply Tango Patterns on switches. One particular basic component is the Switch Inference Engine, which

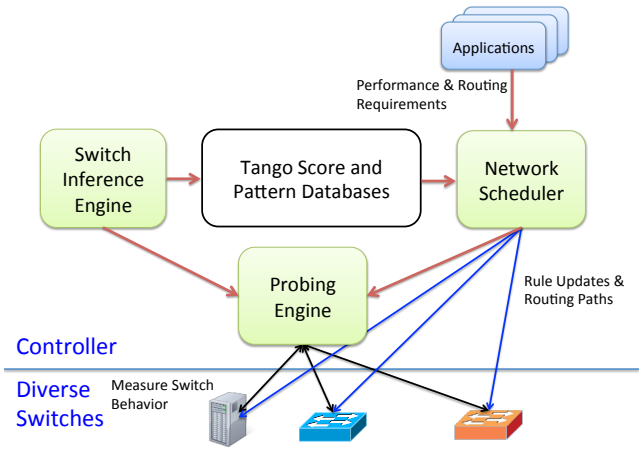


Figure 4: Tango architecture components.

derives switch capabilities and behaviors (e.g., TCAM size, processing throughput, and cache replacement algorithm). The Switch Inference Engine generates Tango Patterns that it may need and stores them in the Tango Pattern Database. It then sends requests to the Probing Engine to apply the patterns. In Section 5, we will define two sets of Tango Patterns which the Switch Inference Engine will use.

Another component that may utilize the Probing Engine and Tango Patterns is Network Schedulers. Our goal is that different Network Schedulers may utilize Tango derived information to achieve its own goals. In Section 6, we will give more details on an example Network Scheduler, which uses Tango derived information to schedule flow priorities and installation sequence, using a technique that is similar to expression rewriting in compiler optimization.

5. HARDWARE INFERENCE PATTERNS

In this section, we design Tango patterns to infer switch sizes and cache replacement algorithms. In particular, we will present two patterns to infer the size and the cache replacement algorithm respectively. The key challenge of designing the two Tango patterns is to handle the challenge that the probing traffic itself can change the cache state. For example, OpenFlow switches keep traffic counters and usage timers that are updated each time the switch receives a packet matching a given flow, so probing a flow that was not initially cached might cause some other flow to be evicted.

5.1 Switch Model

Before explaining our inference algorithms, we first present a conceptual model of an OpenFlow switch that guides our designs. Hardware-based OpenFlow switches (as opposed to Open vSwitch) generally comprise of a flow table that can be partitioned into multiple levels of hardware-backed (TCAM tables) and software-backed (*i.e.*, kernel and user space tables) rules. Since the latency of forwarding using different tables can differ significantly (for instance, hardware forwarding has significantly lower latency than software forwarding), we can view the flow table organization as a multilevel cache for the entire set of forwarding rules in the switch. Analogously, then, the switch policy for deciding how to populate a flow table layer can be viewed as a cache policy. Thus, the flow table layer size, hit rate, and relative performance of hardware and software forwarding of a switch can be used to characterize its performance in the same way as caches in an operating system.

Specifically, consider a cache's managing policy. We formalize such a policy as defining a relation that creates an ordering among all memory elements and removes the element that comes last in this ordering. The cache policy examines some set of attributes of the memory elements (such as usage time or frequency) and then selects the element with the lowest "score" to be evicted (this may be the new element, in which case the cache state does not change). The evicted element may be saved in the next cache, but we focus on one cache at a time. Although many cache policies exist that deviate somewhat from a total ordering (*e.g.*, by introducing randomness), an approximation using a total ordering gives statistically similar results.

Thus, we formalize a switch cache policy as follows:

1. [ATTRIB] The algorithm operates on a subset of the following attributes that OpenFlow switches maintain on a per-flow basis: time since insertion, time since last use, traffic count, and rule priority[1].

2. [MONOTONE] A comparison function between two values of a given attribute is monotonic. The function can be either increasing or decreasing.

3. [LEX] A total ordering on the flows is created by sorting them lexicographically under some permutation of the attributes[2].

We can make a few observations based on these assumptions. First, ATTRIB minimizes the search space of possible cache policies, but in a way that does not affect the generality of the overall probing pattern since both MONOTONE and LEX can apply to arbitrary numbers of attributes. Second, MONOTONE implies that we do not care about the magnitude of the difference between two given values, only the sign. This means that we do not need to worry about the effect of, for example, a single packet on traffic count, provided that the difference between the two flows under comparison was sufficiently large (*i.e.*, greater than 2). Third, LEX is simply a natural extension of single element sort, and is sufficiently general because cache algorithms that do not obey LEX, such as second chance and clock, require hardware support that is not present and unlikely to be built into OpenFlow switches.

Our probing patterns are motivated by the previously mentioned observation that the different flow table layers have significantly different latency. In a two level model of a TCAM-based hardware table and software table, the TCAM-based hardware forwarding has significantly lower latency than software forwarding, so a cached flow entry will have a significantly lower round trip time than one that is not cached. These round trip times will fall into two clusters, with the faster round trip times corresponding to cached entries. Switches with multiple levels of caching (*e.g.*, division of software tables into kernel space and userspace) will have a number of clusters equal to the number of cache layers. This situation is demonstrated in Figure 5, which has three layers. Our algorithm handles the general case of multiple levels of caching.

[1]Note that match layer is likely to be a filter on the cache-eligible flows rather than an ordering constraint.

[2]For example, assume the attributes are permuted in order of traffic count, priority, use time, and finally insertion time. A flow with traffic = 20 would precede one with traffic = 10, and a flow with traffic = 20 and priority = 4 would precede one with traffic = 20 and priority = 2 (the orders are reversed if lower priority or traffic is better).

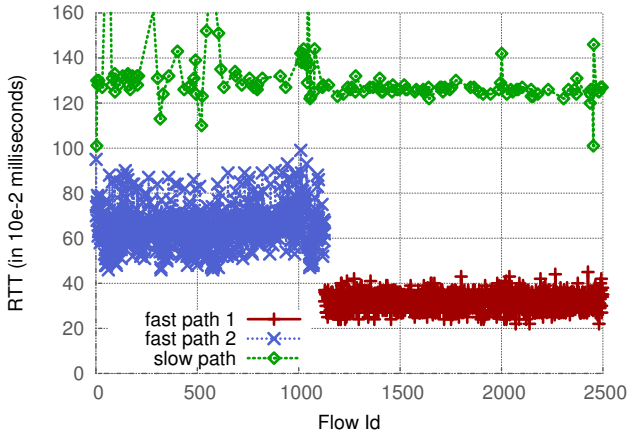


Figure 5: Round trip times for flows installed in HW Switch #2.

5.2 Flow-table Sizes

Our size-probing algorithm (Algorithm 1, Size Probing Algorithm) uses a statistical approach to estimate the flow table sizes of a given switch, relying on the previously described switch model along with the observation that for any possible cache state, a cache hit does not change that state. Thus, we design our algorithm with three stages, as follows. In stage 1 (lines 4-12), as long as we have not overflown the cache (we explain this condition in detail below), we insert a batch of rules (lines 7-10). Stage 2 starts after inserting these rules. In this stage (lines 14-21), we send packets corresponding to the inserted flow rules and measure the RTTs. We cluster the RTT to determine the number of flow table layers– each cluster corresponds to one layer. In stage 3 (lines 23-42), we perform a sampling experiment (let m be the number of rules inserted, and let $n_1, n_2, \ldots, n_N$ be the cache size of levels $1 \ldots N, n = \Sigma n_i$):

1. Select one of the m rules at random (line 27/31)
2. Record the RTT for a packet matching that rule (line 28/32)
3. Repeat the previous two steps as long as the rules are in the same cache layer and we have sent fewer than m packets (line 29)

Since a cache hit does not affect the cache state, this process will terminate if and only if there is a cache miss. Assuming we have $m > n$, therefore, if we record the result of this experiment over multiple trials for a given m, we get an approximation of the expected value of the number of consecutive cache hits. If we perform k trials, the total number of packets sent corresponds to the random variable $X \sim NB(1, n_i/m)$, where $NB(r, p)$ is the Negative Binomial distribution for r negative results sampling with probability p for a cache hit. We can infer the value of p using maximum likelihood estimation over the results of k samples, which yields the expression: $\hat{p} = \frac{\sum_{i=0}^{k} X_i}{k + \sum_{i=0}^{k} X_i}$. Setting $\hat{p} = \hat{n}_i/m$, we get: $\hat{n}_i = m * \hat{p} = m * \frac{\sum_{i=0}^{k} X_i}{k + \sum_{i=0}^{k} X_i}$.

The only remaining question is how we can determine when m has exceeded the total cache size n. By installing flows and sending traffic for each flow upon insertion (before performing the sampling), our switch model guarantees that there are no wasted slots in the cache; that is, there are only empty cache entries if there are fewer flows than slots in the cache. We continue installing new flows until the OpenFlow API rejects the call, which indicates that we have exceeded the total cache size.

Our algorithm is asymptotically optimal in the number of rule installations and probing traffic sent. By doubling the number of rules installed until rule installation fails, and then halving the number of rules and installing, we install n rules $O(n)$ in $2*\log_2 n$ batches, so we can take advantage of batching effects that switches may have for rule installation. Moreover, each rule installed requires 1 packet to be sent. Furthermore, in the experiment phase of the algorithm, we send $O(n)$ packets (we impose a cap on the number of consecutive packets to be m in case n_i is very close to m). Since we are trying to measure the size of the cache layers, we need to install at least n rules and send at least n packets of traffic, so this algorithm being linear with respect to total cache layers' size in both number of rule installations and packets sent is asymptotically optimal.

Algorithm 1 Size Probing Algorithm

```
1:  k ← NUM_TRIALS_PER_ITERATION
2:
3:  function PROBE_SIZE(switch)
4:      cache_full ← FALSE; x ← 1
5:      while !cache_full do
6:          for all i ∈ [0, x), i ∉ switch.flows do
7:              f ← INSERT_FLOW(switch, i)
8:              if rejected then cache_full ← TRUE; break
9:              SEND_PROBE_PACKET(switch, f)
10:         end for
11:         x ← x * 2
12:     end while
13:
14:     g ← ∅
15:     for all i ∈ [0, x) do
16:         f ← SELECT_RANDOM(switch)
17:         rtt ← MEASURE_RTT(switch, f)
18:         APPEND_RTT(g, rtt)
19:     end for
20:
21:     clusters, n_clusters ← COMPUTE_CLUSTERS(g)
22:
23:     for all level ∈ [1, n_clusters) do
24:         a ← 0
25:         for all i ∈ [0, k) do
26:             j ← 0
27:             f ← SELECT_RANDOM(switch)
28:             rtt ← MEASURE_RTT(switch, f)
29:             while WITHIN_CLUSTER(switch, rtt, clusters,
    level) AND j < m do
30:                 j ← j + 1
31:                 f ← SELECT_RANDOM(switch.flows)
32:                 rtt ← MEASURE_RTT(switch, f)
33:             end while
34:
35:             if j = m then
36:                 break
37:             else
38:                 a ← a + j
39:             end if
40:         end for
41:         sizes[i] ← m * a/(k + a)
42:     end for
43:     return sizes
44: end function
```

5.3 Cache Algorithm

We leverage MONOTONE and LEX in order to generate patterns that successfully observe cache state without violating the relative ordering of the inserted flows. LEX suggests a clear structure for

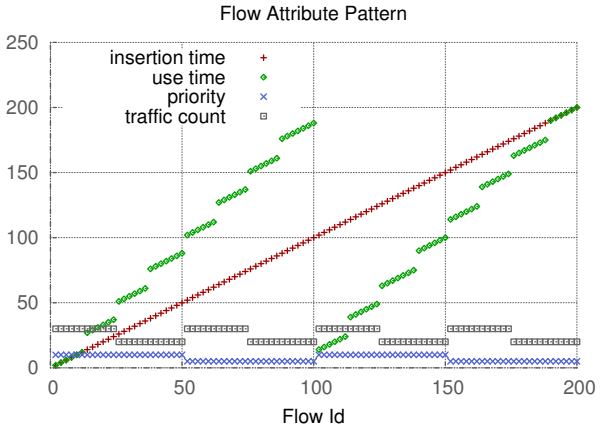


Figure 6: Visualization of Cache Algorithm Pattern for Cache Size = 100.

our probing. If we first identify the primary sorting attribute, we can then hold it constant to recursively identify the next sorting attribute, and continue until we have determined the correct permutation of the attributes. We identify the primary sorting attribute by ensuring, for each attribute for a given flow, the corresponding value is either less than or greater than the values of half (n = cache size) of the other flows. If we ensure that there is no subset of flows for which this condition holds for more than one attribute at a time, we can fairly easily identify the sorting attribute by seeing which correlates most strongly (positive or negative) with the caching decisions.

Algorithm 2 presents the details of policy patterns. We first insert $2 * cache\_size$ flows (line 4) and initialize them subject to the constraints explained in the paragraph above (lines 5-7). The traffic generation pattern in line 9 ensures that our probing does not affect the flow's relative position in an ordering on any attribute. Considering each attribute in turn, priority and insertion time are unaffected by probing, whereas traffic count could be affected by probing if two flows had counts that differed by only one, but the algorithm initializes the values to be far apart (10 in our specific implementation). We maintain invariant relative use times by querying in order of reverse use time (line 9). For example, by the time we send traffic for what was initialized as fifth most recently used rules, only the four rules that had succeeded it have been queried again, so the ordering is left unchanged.

Similarly to the size probing patterns, the policy patterns examine the round trip times to determine which elements are cached (based on the data collected during size probing). LEX guarantees that the cached flows should have some attribute for which they all have the highest values or all have the lowest ('highest' and 'lowest' meaning greater than or less than the values for at least n other flows). We identify this variable by statistical correlation (line 24), and then recur to find subsequent sort attributes if necessary (lines 27-30).

As an example, consider probing the cache replacement algorithm for a cache of size 100. The post-initialization state of each of the flows is shown in Figure 6. If the replacement algorithm is LRU, we will see that packets matching the flow entries with $use\_time > 100$ will have round trip times near the mean cache processing time computed as part of the size inference process. Since use time values are unique, the flow entries can be arranged in a strict total order on the basis of just use time, and the algorithm would terminate.

Algorithm 2 Policy Probing Algorithm

```
1:  ATTRIBUTES ← {insertion, usage_time, traffic, priority}
2:  SERIAL_ATTRIBUTES ← {insertion, usage_time}
3:
4:  function PROBE_POLICY(switch, policy)
5:      attributes ← ATTRIBUTES \ policy
6:      s ← 2 * switch.cache_size
7:
8:      for all i ∈ [0, s) do
9:          f ← INSERT_FLOW(switch, i)
10:         for all a ∈ attributes do
11:             INITIALIZE_FLOW_ATTRIBUTE(switch, f, a)
12:         end for
13:     end for
14:
15:     rtts ← {}
16:     for all f ∈ SORT_BY_MRU(switch.flows) do
17:         rtts[f] ← MEASURE_RTT(switch, f)
18:     end for
19:
20:     correlations ← {}
21:     for all o ∈ ATTRIBUTES × {incr, decr} do
22:         correlations[o] ← CORRELATE(o, rtts)
23:     end for
24:     a ← ARGMAX(correlations)
25:     policy.APPEND(a)
26:
27:     if a ∈ SERIAL_ATTRIBUTES then
28:         return policy
29:     else
30:         return PROBE_POLICY(switch, policy)
31:     end if
32: end function
```

6. TANGO SCHEDULER

The hardware table inference patterns provide basic information about individual switches and their performance. Rewriting patterns provide information to allow a scheduler to identify effective operational patterns. In this section, we demonstrate the benefits of using Tango pattern in scheduling and optimization of flow installation. We emphasize that the algorithm depends on the available patterns, and designers should continue to add new patterns to guide scheduling and optimizations.

Application requests: The design intention of Tango is that it can be integrated with a wide range of SDN controllers, and different SDN controllers may offer different APIs for applications to submit requests. Hence, application requests can range from simple static flow pusher style requests (*e.g.*, [20]), where the whole path is given in each request, to declarative-level requests such that the match condition is given but the path is not given (*e.g.*, ACL style spec) in each request [14], to algorithmic policies such that even the match condition will be extracted (*e.g.*, [23]).

Switch request DAG: Given application requests and/or other events (*e.g.*, link status changes), a compiler or run time at the central SDN controller will generate operations, which we call switch requests, to individual switches to update the rules in their flow tables. Using a general format, we consider that each request has the following format:

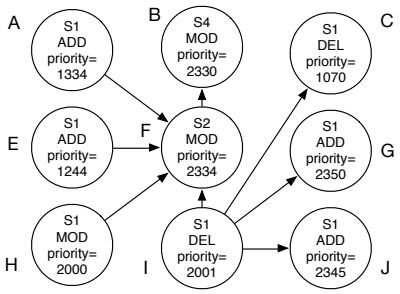


Figure 7: An example DAG of switch requests.

```
req_elem = {
     'location': switch_id,
     'type' : add | del | mod,
     'priority': priority number or none
     'rule parameters' : match, action
     'install_by': ms or best effort}
```

Each switch request has a location field, which specifies the switch at which the operation should be installed. A switch request also has other common fields such as the operation type, the priority of the new flow table rule, and other parameters such as match fields and actions. Priority is needed, since that TCAM match conditions cannot include negation, and hence one must introduce higher-priority barrier rules [23] for such cases. A switch request may have a deadline by which the request should be installed (*e.g.*, derived from aforementioned application requests). A switch request may leave the deadline field empty for best efforts.

Switch requests may not be independent. For example, to achieve consistent updating, a controller may specify that the requests are finished in a given order (*e.g.*, from the destination switch back to the source switch, for the path of a new flow). The aforementioned case of using priority to implement logic negation may also require that some rules are installed before the other, to avoid transient breaches. Using a graph G where each node represents a switch request, if a switch request A must be completed before another request B, we say that there is a direct edge from A to B in the graph. Our Tango scheduler assumes that the graph G is a directed-acyclic graph (DAG). If the dependency forms a loop, the upper layer must break the loop to make G a DAG. Figure 7 shows an example DAG that indicates multi-switch requests, and their dependencies. At any instance of time, we say that the controller has a current switch-request DAG.

Basic Tango scheduler: The basic Tango scheduler algorithm, shown in Algorithm 3, processes the switch requests in the current switch-request DAG. To understand Algorithm 3, first consider the scheduling of switch requests without Tango measurements. Given the current DAG, it is intuitive that one schedules the requests starting from those with no dependencies, such as the Dionysus [6] algorithm. In particular, the algorithm identifies those requests without dependencies, which we refer to as independent requests, and schedules the independent request that belongs to the longest path first.

A major issue of this approach, however, is that a typical DAG can have a large number of independent requests with the same longest-path length. For example, consider Figure 7. The three requests on the left (i.e., requests A, E, H) and the one on the lower middle (i.e., request I) have no dependency and have the same longest path length. Hence, a key issue is how one schedules these four requests. This is where Tango patterns may provide substantial guidelines. Specifically, at line 3, the scheduler extracts the current independent set of switch requests (*e.g.*, during the first round, the requests A, E, H, I), and uses the Tango Score and Pattern Database for a suggested ordering of the switch requests in the set. Applying pattern 1 (deletions followed by modifications followed by additions in ascending priority order) at line 12, the algorithm computes a score of -91 ($= -(10 \times 1 + 1 \times 1 + 20 \times 2^2)$, where the first 1 means one DEL, the second 1 means one MOD in the set of requests, and 2 for the 2 ADD requests, and 10, 1, and 20 are the weights for DEL, MOD, and ADD, respectively). Applying pattern 2 gives a lower score of -171. Hence, the scheduler picks the first scheduling pattern and issues the requests in the suggested order.

Algorithm 3 BasicTangoScheduler

```
 1: function BASICTANGOSCHEDULER(G)
 2:     while true do
 3:         G_i ← independent set in G
 4:         Git ← orderingTangoOracle(G_i)
 5:         issue requests in Git
 6:         remove finished requests in Git
 7:     end while
 8: end function
 9: function orderignTangoOracle(G_i)
10:     maxScore ← NEGATIVE_INFINITY
11:     for all pattern ∈ TangoPatterns do
12:         score ← computePatternScore(G_i, pattern)
13:         if (score > maxScore) then
14:             maxScore ← score
15:             maxPattern ← applyPattern(Gi, pattern)
16:         end if
17:     end for
18:     return maxPattern
19: end function
20: TangoPatterns ← [
21: {pattern : "DEL MOD ASCEND_ADD",
22:  score : −(10 × {DEL} + 1 × {MOD} + 20 × {ADD^2})},
23: {pattern : "DEL MOD DESCEND_ADD",
24:  score : −(10 × {DEL} + 1 × {MOD} + 40 × {ADD^2})},
25: ...
26: ]
```

Extensions: The basic scheduling algorithm uses greedy batching of independent requests. For example, the basic algorithm schedules all four requests (A, E, H, I) in one batch, in the order of I, H, E, A. One extension that may improve the basic algorithm is to consider non-greedy batching. Specifically, after computing the ordering I, H, E, A, the scheduler evaluates different prefixes to determine if there is a better ordering, by considering issuing the prefix, and then the batch enabled by the completion of the prefix. Consider the prefix I, its completion will produce a new independent set (A, E, H, C, G, J). The scheduler computes the total score of schedule (1): batch 1 with I, and batch 2 with C, H, E, A, J, G, vs that of schedule (2): batch 1 with I, H, E, A and then batch 2 with C, F, J, G. This process leads to a scheduling tree of possibilities and the scheduler picks the schedule with the best overall score.

Another extension of the basic algorithm is to schedule dependent switch requests concurrently. Specifically, Tango probing engine can provide measurement results to guide the scheduling of concurrent requests. Specifically, in case of a dependency S_1 ADD$\rightarrow$ S_2 ADD (between switches S_1 and S_2), the scheduler does not have to wait for the first command to finish if we can estimate (using the Tango latency curves) that the second command is much slower than the first (e.g. if the switch is overloaded, or it has a

Flow Files	Topological Priorities	R Priorities	Flows Installed
Classbench1	64	829	829
Classbench2	38	989	989
Classbench3	33	972	972

Table 2: Number of flows per Classbench file and their associated priorities.

slower CPU). On the contrary, we can push both rules such that the estimated (using the Tango latency curves) finishing time of the operation S_1 ADD precedes the estimated finishing time of the rule S_2 ADD by a "guard" time interval. This is suitable for senarios with weak consistency requirement.

7. EVALUATIONS

In this section, we present the results from the evaluation of Tango and demonstrate that using Tango patterns, we can achieve substantially better SDN programming efficiency both in a per-switch and network-wide basis.

7.1 Single Switch Evaluation

With basic switch properties inferred from the preceding section and additional rewriting patterns, Tango scheduler optimizes the assignment and installation of flow table rules. We evaluate how much gain that Tango can achieve compared with approaches that are oblivious of switch diversity. For this, we test four algorithms that combine different priority assignments and installation order to the rules and measure their performance.

Methodology: In order to find the optimal scheduling policy that can minimize flow installation time, we use three sets of rules from Classbench [21] access control lists to generate flow rules with dependencies. We then process the dependency graph with the algorithm taken from [23] in order to generate a) the minimum set of priorities needed to install the rules while satisfying the dependency constraints, and b) a set of priorities that are mapped 1-1 to the flow rules but they still satisfy the dependency constraints of the rules.

We denote the first set of priorities *Topological Priorities* (we use topological sorting to obtain the rule schedule and then use the same priority number for rules that have no dependencies with each other), and the second R Priorities. The next step is to measure the installation time using Tango pattern under the following scheduling scenarios: 1) topological priority assignment and the optimal rule installation order taken from our probing engine, 2) topological priority assignment and random rule installation order, 3) 1-1 priority assignment that satisfies the dependency constraints and the optimal installation order taken from our probing engine, and 4) 1-1 priority assignment that satisfies the dependency constraints and random installation order. We repeat the above experiment three times using three different Classbench files. The number of flows per Classbench file as well as the number of Topological and R priorities generated are shown in Table 2.

Per Switch Evaluation: For each Classbench file, and for the four scheduling scenarios shown above, we measure the installation time in OVS and HW Switch #1, as shown in Figures 8(a)-9(c). We can observe that the topological priority assignment combined with the optimal policy taken from our probing engine achieved the best performance in terms of flow installation time in five out of the six scenarios under study. For example, in Figure 8(a)-8(c), we can observe that the decrease in the installation time is 10%, 9%, and 8%, respectively. In addition, in the case of the hardware switch (Figure 9(a)-9(c)), the decrease in the installation time is 87%, 80%, and 89%, respectively. The improvement in the case of OVS is lower since OVS is very fast for a relatively small number of rules (i.e., <1000).

The only exception was the case of Classbench 1 for HW Switch #1, where the installation time is slightly more than the one using priority assignment of type R and the rules are installed in ascending priority.

To further analyze the behavior of real hardware switches, we repeated the experiment using descending priority installation order, instead of ascending. As expected, the installation time was up to 12 times higher compared to the one using ascending priority (we do not show these results here due to space limitations).

7.2 Network-Wide Evaluation

To further evaluate the performance of Tango, we conduct experiments on both a hardware testbed and a virtual network emulator (Mininet [24]), that represent the following real-life scenarios: (a) *Link Failure (LF):* one or more links from the physical topology fails causing existing flows to be rerouted; and (b) *Traffic Engineering (TE):* a traffic matrix change causes a number of flows to be added, removed, or modified. In both scenarios, we ensure that the flow updates are conducted in reverse order across the source-destination paths to ensure update consistency [18].

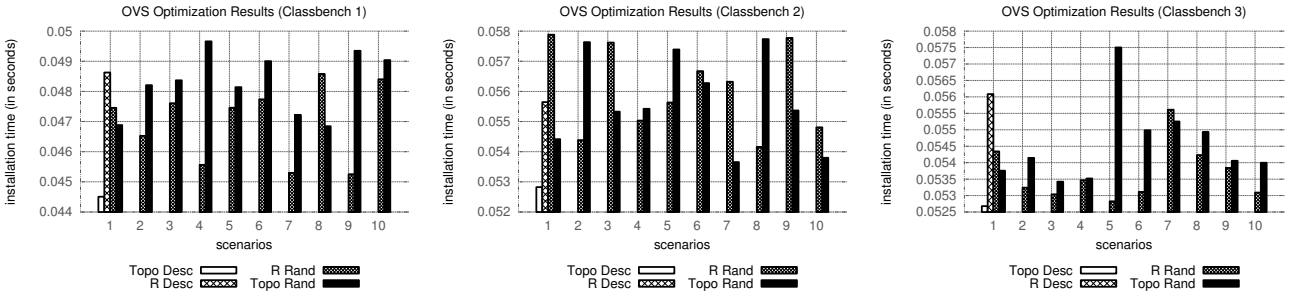
We compare the performance of Basic Tango Scheduler with Dionysus [6], a network update scheduler for SDN. The core of Dionysus uses a critical path scheduling algorithm that constantly calculates the critical path on the scheduling DAG (that represents the rules to be pushed to each switch and their dependencies), and pushes the rules to the switches that belong to the critical path, first. However, unlike Tango, Dionysus does not take into account the diversity in the latency of each rule type, as well as the diversity of the switch implementations (e.g., caching algorithm).

Evaluation on hardware testbed: First, we conduct experiments on a small testbed consisting of three hardware switches, denoted as s_1, s_2, and s_3. Switch s_1 and s_2 are from Vendor #1, and s_3 is from Vendor #3. Initially these switches are fully connected forming a triangle. We use this testbed to implement both LF and TE scenarios, as follows. In the LF case, the link between s_1 and s_2 got disconnected causing 400 existing flows to be rerouted from s_1 to s_2 via s_3. In the TF case, a traffic matrix change at s_1 causes a number of flows to be added, removed, or modified. We have implemented two TE scenarios (denoted as TE 1 and TE 2) with different distributions on the number of rules of each type required (i.e., ADD, MOD, DEL) in order to transition from the initial state to the target state. In the TE1 scenario, we randomly generate 800 flow requests with twice as many additions as deletions or modifications, while in TE2, these three types of requests are equally distributed. We assume explicitly specifies priorities for their rules, Tango can optimize the rules with priority pattern (called *Priority sorting*).

Figure 10 shows the results of LF, TE1, and TE2 scenarios. By applying only a rule-type optimal pattern, Tango gets 0%[3], 20%, and 26% improvement compared to Dionysus, respectively. In addition, by applying both rule-type pattern optimization and priority pattern optimization, Tango could further reduce the flow installation time by 70%, 33% and 28%, respectively.

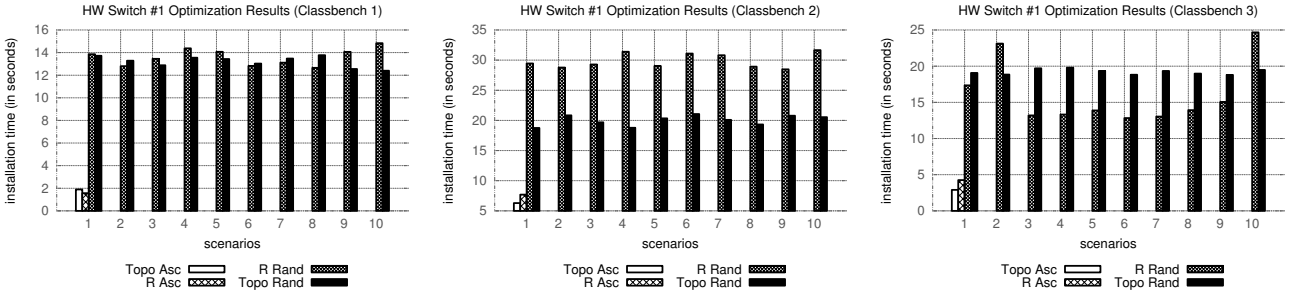
Figure 11 explores two settings about priorities: The first is *priority sorting* as described above. The second is *priority enforcement*. When applications do not specify the flow priorities (as men-

[3]This is because in LF scenario, there are only rule additions on s_1 and rule modifications on s_2. Thus, there is no room for rule-type optimization.



(a) Classbench 1 (b) Classbench 2 (c) Classbench 3

Figure 8: Installation time for various priority assignments and installation order in OVS.



(a) Classbench 1 (b) Classbench 2 (c) Classbench 3

Figure 9: Installation time for various priority assignments and installation orders in HW switch #1.

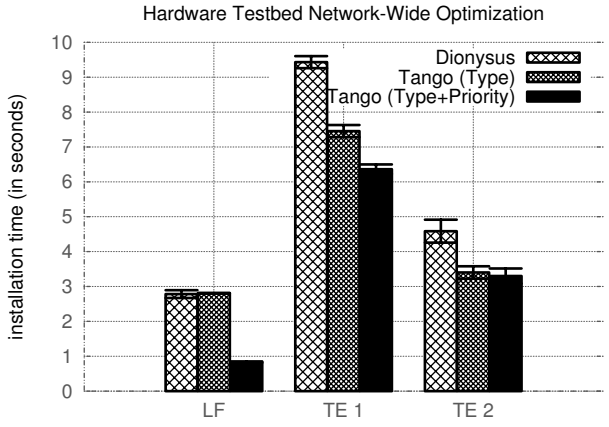


Figure 10: Hardware testbed Link Failure and Traffic Engineering Scenarios.

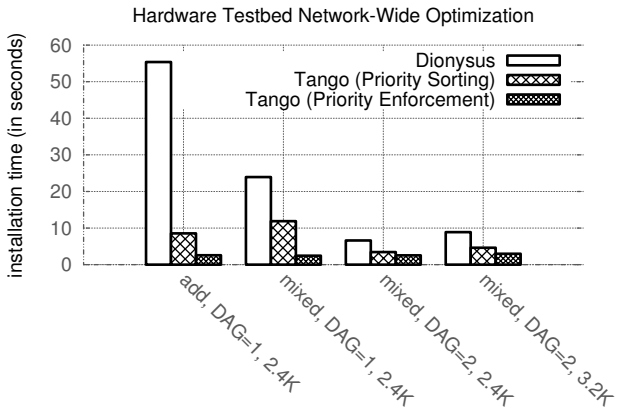


Figure 11: Hardware testbed measurements.

tioned in Sec. 6), Tango can optimize the assignment of priorities based on the dependency constraints. We evaluate four scenarios, each with a different type of rules pushed in the network (i.e., ADD only, or mix of ADD, MOD, DEL), a different number of levels in the dependency graph, and a different total number of rules. In general, Tango outperforms Dionysus in all cases with a maximum improvement of 85% and 95% via priority sorting and priority enforcement, respectively. This happens once the request only consists of flow additions. In the last two scenarios, we explore the situation where the DAG has more than one levels, under the same number of rules as before. In these cases, the performance benefit of Tango is smaller, since each level has less independent rules, and thus less optimization options.

Network emulation on OVS: To evaluate Tango in larger networks, we evaluate Tango scheduler on Google's backbone network topology [5] using Mininet [24]. In the TE case, we implement a traffic matrix change in the B4 topology that triggers a sequence of rule additions, modifications, and deletions, based on the max-min fair allocation algorithm used in [5]. The results are shown in Figure 12. We observe 8% flow installation improvement compared with Dionysus results, for a total of 2200 end-to-end flow requests. Similar improvements are observed in the LF case. This improvements come from rule-type pattern. This is smaller than the improvements on hardware testbed, because OVS has smaller performance differences betwen rule type patterns. We also note

that priority optimization does not help in the case of OVS. This is priority has little impact on OVS rule installation performance.

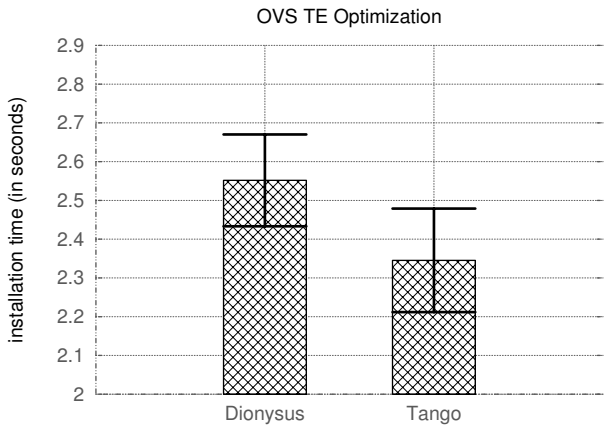


Figure 12: Mininet TE Scenario.

8. RELATED WORK

Managing switch diversity: Discussions on OpenFlow 2.0 (*e.g.*, [1]) highlight the difficulties of handling switch diversity. OF-config [15] provides a management protocol for the controller to understand and control switch-specific configurations. However, these protocols do not fully capture switch diversity and do not provide mechanisms to quantify the diversity. For example, they do not specify a mechanism to query the caching policy, and quantify switch update latencies. In [4], the authors explore the diversity of different switch implementations using manual inspection. We propose algorithms to automatically infer switch parameters and Tango patterns. Oflops [19] provides a standalone controller to benchmark OpenFlow switch performance. Tango builds on Oflops but designs smart probing algorithms to infer Tango patterns.

High level abstractions: Existing SDN controllers and languages (e.g., Frenetic [12], Onix [8], Maple [23]) provides a high-level abstraction of network elements (e.g., switches, hosts, ports) to reduce the complexity of programming SDN, and leverage compilers and runtime systems to optimize the application performance independent of diverse switches. In Tango, we optimizes application performance by leveraging switch-specific opportunities (i.e., Tango patterns). One can view Tango as a "backend" compiler optimizer which none of the existing compilers [3, 23] provide. It is possible to integrate Tango with Frenetic and Maple. We leave that to future work.

Switch level optimizations: Previous work has proposed optimizations at both software and hardware switches to improve performance. For example, DevoFlow [2] introduces rule cloning to reduce the use of TCAM entries. CacheFlow [7] proposes new caching solutions for overlapping rules between software and hardware switches. However, it is challenging for applications to leverage these optimizations without knowing their design details. With Tango, these optimizations are just new Tango patterns for applications.

There have also been active research on optimize rule updates to ensure consistency [18, 10] and reducing transient congestions during updates [9, 6]. Complementary to these works, we focus on how to reorder or modify rule updates based on diverse switch capabilities to improve performance. One can view Tango as a mechanism to optimize per-switch rule updates. The consistent update mechanisms can use Tango to minimize their overall rule setup completion time.

9. CONCLUSION AND FUTURE WORK

Our measurement studies show that SDN switches have diverse implementations, capabilities and behaviors. Current controller implementations ignore the diversity. This leads to inefficient usage of switch resources. Most importantly, ignoring switch diversity can adversely impact application performance, e.g. packets end up in switch slow path. To simplify SDN programming, we present Tango, the first systematic design. Tango automatically infers switch behavior using a well-structured set of Tango patterns, where a Tango pattern consists of a sequence of standard OpenFlow commands and a corresponding data traffic pattern. Tango presents a simple API for controller applications to specify their requirements. Tango optimizes network performance exploiting the cost of a set of equivalent operations through expression rewriting. Our evaluation shows the effectiveness of Tango. For our future work, we would like to expand the set of Tango patterns to infer other switch capabilities such as multiple tables and their priorities.

Acknowledgments

We thank Haibin Song for providing examples of specific switches; James Aspnes and Shenil Dodhia for extensive discussions on the inference algorithms; and CoNEXT reviewers for their suggestions. Daniel Tahara is partially supported by a gift from Futurewei. Andreas Voellmy and Y. Richard Yang are supported by a gift from Futurewei and NSF CNS-1218457. Minlan Yu is supported in part by NSF CNS-1423505.

10. REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communications Review*, 2013.

[2] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of ACM SIGCOMM*, August 2011.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2011.

[4] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, August 2013.

[5] 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 *Proceedings of ACM SIGCOMM*, August 2013.

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

[7] N. Katta, J. Rexford, and D. Walker. Infinite CacheFlow in software-defined networks. *Princeton Computer Science Technical Report TR-966-13*, 2013.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, October 2010.

[9] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM*, August 2013.

[10] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets-XII)*, November 2013.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communications Review*, 2008.

[12] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2013.

[13] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[14] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2014.

[15] OF-config: `https://github.com/AndreasVoellmy/data-driven-sdn-paper`.

[16] OpenFlow switch specification 1.4.0: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf`.

[17] Openvswitch: `http://openvswitch.org/`.

[18] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets-X)*, November 2011.

[19] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for OpenFlow switch evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, March 2012.

[20] Ryu SDN controller: `http://osrg.github.io/ryu/`.

[21] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, pages 499–511, 2007.

[22] The eXtensible DataPath Daemon (xdpd): `https://www.codebasin.net/redmine/projects/xdpd/wiki`.

[23] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM*, August 2013.

[24] Mininet: An Instant Virtual Network on your Laptop. `http://mininet.org`.