Composing SDN Controller Enhancements with Mozart

Zhenyu Zhou
Duke University
zzy@cs.duke.edu

Theophilus A. Benson
Brown University
tab@cs.brown.edu

ABSTRACT

Over the last few years, we have experienced a massive transformation of the Software Defined Networking ecosystem with the development of SDNEnhancements, e.g., Statesman, ESPRES, Pane, and Pyretic, to provide better composability, better utilization of TCAM, consistent network updates, or congestion free updates. The end-result of this organic evolution is a disconnect between the SDN applications and the data-plane. A disconnect which can impact an SDN application's performance and efficacy.

In this paper, we present the first systematic study of the interactions between SDNEnhancements and SDN applications – we show that an SDN application's performance can be significantly impacted by these SDNEnhancements: for example, we observed that the efficiency of a traffic engineering SDN application was reduced by 24.8%. Motivated by these insights, we present, Mozart, a redesigned SDN controller centered around mitigating and reducing the impact of these SDNEnhancements. Using two prototypes interoperating with seven SDN applications and two SDNEnhancements, we demonstrate that our abstractions require minimal changes and can restore an SDN application's performance. We analyzed Mozart's scalability and overhead using large scale simulations of modern cloud networks and observed them to be negligible.

CCS CONCEPTS

• **Networks → Programmable networks**; **Network management**.

KEYWORDS

Software Defined Networks, Composition, Compilers

ACM Reference Format:
Zhenyu Zhou and Theophilus A. Benson. 2019. Composing SDN Controller Enhancements with Mozart. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362712

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '19, November 20-23, Santa Cruz, CA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362712

1 INTRODUCTION

"The art of simplicity is a puzzle of complexity."
—Douglas Horton.

Cloud providers employ Software Defined Networking (SDN) to simplify network management and amongst other things to expedite virtual network provisioning [13, 15, 16, 30]. With SDNs, providers can now configure their networking infrastructure using higher level abstractions provided by SDN Applications (SDNApps) rather than through low-level commands provided by device vendors.

To enable innovation, SDN-developers often decouple the creation and design of individual networking functionality (encapsulated in SDNApps) from global network-wide optimizations (encapsulated in SDNEnhancements). Unlike SDNApps which provide specific network functionality (e.g., traffic engineering or network virtualization), SDNEnhancements are designed to address deficiencies in the SDN ecosystem and provide general optimizations for SDNApps (e.g., better utilization of TCAM; consistent network updates – a more exhaustive list is provided in Table 1).

Class of SDNEnhancement	Example	Description
Conflict-Resolver	[14, 48]	Enforces resource allocation to different SDNApps
TCAM-Optimizer	[25, 51]	Minimizes switch memory (TCAM) utilization
Consistent Update	[33, 40, 45]	Updates network paths in a consistent manner
Invariant Checker	[27, 28]	Checks to see if a network invariant holds (e.g. no cycles)
SDNApp Composition	[5, 35, 37]	Combines rules from different SDNApps
Fault Tolerance Path	[44]	Automatically creates backup paths to overcome link failure

Table 1: Taxonomy of SDNEnhancements.

These SDNEnhancements have evolved organically in response to the recent issues network administrators faced while deploying SDNs. For example, the controller's inability to perform congestion-free network updates [33, 45] which results in network performance anomalies or deficiencies within the data-plane update mechanisms, e.i., consistent update problems [45] (Section 2).

As a result of this organic evolution, today many SDNEnhancements have adhoc designs. In particular, SDNEnhancements are either co-designed with SDNApps which limits their generality or SDNEnhancements are inserted transparently into the SDN ecosystem which, while improving generality, hurts the SDNApp's performance. The latter impacts performance because it creates a disconnect between the SDNApp's view of the network and the actual network state: a disconnect between the control messages (forwarding rules) generated by an SDNApp and the forwarding rules stored in the data-plane which can impact an SDNApp's performance by as much as 28% (Section 3).

In this paper, we take a step back and ask more fundamental questions:

"What is the right interface for enabling principled interactions between SDNApps and SDNEnhancements? What abstractions are required to systematically include SDNEnhancements into the SDN ecosystem?"

To answer these questions, we take inspiration from the compiler community and their toolchain design where (1) compiler optimizations are *explicitly* configured by a developer, (2) *flags* are used to express *hints* that ensure that the optimizations do not *impact program intent*, and (3) optimizations are treated as transformations on an intermediate representation which allows for more systematic reasoning of their implications. Motivated by these insights, we argue for designing an *intermediate representation* of the SDNApp control messages, a representation that is amendable to both principled analysis and modifications by SDNEnhancements. Furthermore, we argue that SDNEnhancements should be more systematically included into the SDN environment but treated as black box transformation engines that operate on intermediate representation and create intermediate representation as output. Given this model, administrators can control transformations with SDN-Flags.

Current solutions to SDN composition fail to answer our original questions. First, traditional SDNApp composition (e.g. Pyretic [37]) focuses on safely combining multiple SDNApps and tackling the complexity arising from sharing network resources. Instead, we focus on the SDNEnhancements applied to the resulting composed rules. Second, novel interfaces between the SDNApp and SDNEnhancements, e.g., Athens [5], require the SDNApp developers to write code that analyzes and evaluates the transformations made by SDNEnhancements. Unfortunately, this interface requires the SDNApp to understand the implications of all potential SDNEnhancements. We argue that developers should simply specify the class of transformations that are tolerable, or not, without needing to understand or evaluate the multitude of SDNEnhancements (or their combined transformations).

In this paper, we propose Mozart, a novel controller framework that introduces, a simple but powerful interface that standardizes interactions between controllers and the SDNEnhancements thus enabling us to systematically reason about SDNEnhancements: to mitigate the implications of SDNEnhancements on SDNApps we propose a set of SDN-Flags, akin to compiler flags, that lets SDNApps specify the class of transformations that impact correctness or efficiency. While we have implemented our abstractions with two popular controllers, we believe that our abstractions can be easily incorporated into emerging research prototypes, e.g., SoL [18] and YANC [36].

In summary, we make the following contributions:

- **Systematic Study of Complexity:** We present a systematic study of the implications of applying realistic SDNEnhancements to SDNApps and show that an SDNApp's performance can be reduced by as much as 24.8% (Section 3).
- **SDN Abstractions:** We describe a set of interfaces and abstractions for mitigating and reducing the impact of these SDNEnhancements on SDNApps (Section 4).
- **Implementation & Evaluation:** We build a working prototype implementation of Mozart on two controllers (Floodlight [2] and Ryu [1]) and demonstrate the benefits of our primitives with seven SDNApps and two SDNEnhancements (Section 7). Our evaluations demonstrate that our prototype can minimize the impact of these SDNEnhancements. Moreover, we show that our abstractions are non-invasive and require as little as 18 lines of code changes to the SDNApps (Section 7).

Roadmap. In Section 2, we describe the structure of modern SDNApps and highlight problems in SDNEnhancements. Then, in Section 3, we study the implications of applying SDNEnhancements to SDNApps. In Sections 4 and 5, we present our abstractions and models. In Sections 6 and 7, we present our prototype and its evaluation. We present discussions and related works in Section 8 and 9. Section 10 concludes with final remarks.

2 MOTIVATION

In this section, we describe the fundamental structure of an SDNApp, present the simplifying assumptions that SDNApps make about the networks, and conclude by discussing a subset of SDNEnhancements that have been developed to correct the implications of these assumptions.

2.1 The Case for SDNEnhancements

SDNApps encapsulate control-plane functionality (network policies) and are designed to be event-driven. They interact with the data-plane by generating SDN control messages, e.g., OpenFlow messages (forwarding rules). We illustrate the need for SDNEnhancements by examining a canonical traffic engineering SDNApp, e.g., Hedera [4], and analyzing its interactions with the network. Hedera, Algorithm 1, aims to improve data center performance by detecting elephant flows and load balancing them on distinct paths. Hedera does this in three steps: (1) monitoring the network and collecting statistics, (2) detecting elephant flows and calculating new paths to ensure load is balanced, and (3) configuring new paths into the network with OpenFlow control messages.

SDNApps are written using one of two well-established patterns: proactive [4, 8, 9, 20, 22] and reactive [41, 42]. The fundamental difference between the two patterns is that the event loops for proactive SDNApps, e.g., Hedera, is triggered by a timer whereas reactive SDNApps are triggered purely by the arrival of network events, e.g., Packet-In events. The discussion below applies equally to both classes of SDNApps. In applying these control messages to the network, SDNApps, including Hedera, make the following assumptions about the network:

Instantaneous Updates: SDNApps assume that the SDN controllers instantaneously apply OpenFlow rules to the network devices. However, network latency between the controller and devices leads to out of order or delayed updates. A class of SDNEnhancements [33, 45], *Consistent-Update*, have been developed to ensure atomic and consistent updates.

```
while true do
    /* Get Network Input */
    foreach device in Network do
        Counters.Append(device.GetStatistics())
    end
    /* Control Function */
    Rules = BinPackingHeuristic(Counters)
    /* Send Output to Network */
    foreach device in Network do
        device.installRules(Rules)
    end
    Sleep100msecs
end
```

Algorithm 1: Pseudocode for Hedera, An SDN Application for Traffic Engineering in Data Centers.

Implication on SDNApps: The SDNEnhancements introduce consistency by employing techniques motivated by 2-phase commit or causal consistency. The implication of these SDNEnhancements is a temporary *duplication* of rules: the old and the new. This essentially transforms the OpenFlow-message into two duplicate messages. Unfortunately, the SDNApps are unaware of the old rules and will subsequently ignore them and their associated metadata. For example, Hedera installs rules as output but also collect the metadata from these rules as input. Unfortunately, Hedera will only ask for metadata for the rules it is aware of – assuming that the old rules have been deleted the Hedera will ignore them. Lacking such metadata may reduce the *efficiency* or *accuracy* of the control functions of SDNApps such as Hedera.

Infinite Hardware Resources: SDNApps assume an infinite amount of device memory (TCAM); However, TCAM space is limited in existing switches. Most can support $\sim 1K$ rules. The design choice of abstracting out details and limitations of the physical hardware is a common system design principles (e.g., an OS provides virtual memory). However, unlike an operating system which provides adequate abstractions to support this, an SDN controller does not. Thus to overcome this limitation, a class of SDNEnhancements [25, 51], *TCAM-Optimizers*, have been developed to provide the illusion of infinite memory.

Impact on SDNApps: These SDNEnhancements create optimized-rules that efficiently utilize switch TCAM by merging, moving or splitting the rules generated by the SDNApp: essentially **transforming** an OpenFlow-message into *Coarser* Granularity or *Finer* Granularity messages. Unfortunately, certain SDNApps install rules of a certain granularity under the **assumption** that these rules can be used to collect metadata of flows at the pre-specified granularity. The implication of these coarser granularity rules is that metadata can only be collected at that coarser granularity. For Hedera, a direct implication is that the control function may be unable to load-balance at a finer-granularity thus impacting Hedera's *effectiveness* (we empirically quantify this impact in Section 3).

Unmodified Actions: SDNApps assume that the network receives and faithfully enforces the actions associated with the rules it installs.

Impact on SDNApps: In addition to modifying an OpenFlow-rule's match by making it coarser or finer, SDNEnhancements may also change the OpenFlow-rule's actions. For example, DiFane [51], a TCAM optimizing SDNEnhancement alters paths and uses detours to minimize the number of TCAM entries. In general, SDNEnhancements may transform actions in one of the following ways: (1) changing the network path by altering the interface associated with an action, (2) changing the reachability by changing the action, or (3) changing the QoS disciplines by changing the queues associated with the action. For Hedera, a direct implication of path changes (detours) is that large flows explicitly being isolated may be placed on identical links resulting in congestion. This would minimize Hedera's effectiveness.

2.2 SDNEnhancement Definition

An SDNEnhancement is a controller add-on which augments controller's base functionality by providing additional properties to the applications beyond simple demultiplexing and multiplexing of the control messages. Given this definition, the fundamental distinction between SDNApps and SDNEnhancements lies in where network control and management policies lie. The SDNApps encapsulate and contain the management policies – the OpenFlow messages that they generate reflect these policies. On the other hand, SDNEnhancements take in the OpenFlow rules (or policies) created by SDNApps and perform some optimizations (e.g., TCAM optimizers) or sanity checks (e.g., conflict resolvers or consistent updates). In general, SDNEnhancements do not themselves contain any network policies and by themselves. In short, SDNEnhancements cannot run or control the network.

2.3 SDNEnhancement Deployment Scenarios

These SDNEnhancements are often bundled as a part of the controller and in a few cases they are deployed as a proxy service between the controller and the data-plane. In both situations, the SDNEnhancements and the transformations that they perform are hidden from the SDNApps.

Takeaways. Current SDN controllers lack appropriate primitives to enable higher level SDNApps to efficiently and safely utilize switch's hardware. While many SDNEnhancements have been developed to provide these primitives to SDNApps, transparently applying SDNEnhancements to unsuspecting SDNApps can result in disastrous consequences, e.g., correctness violations, compromised accuracy, or reduced reactiveness. In this section, we present a representative set of SDNEnhancements and SDNApps and use them to illustrate the dangers of naively interposing SDNEnhancements between SDNApps and the data-plane.

Moreover, our observations extend to other SDNEnhancements not discussed here, such as, Invariant-Checkers [27, 28], which have similar problems as Conflict-Resolver SDNEnhancements [14, 48].

3 UNDERSTANDING SDN-ENHANCEMENT

We now present empirical data to quantify the impact of SDNEnhancements on SDNApps: we focus on the TE-SDNApp discussed in Section 2 (Hedera) and analyze reduction in aggregate bandwidth (*efficiency*) which allows us to understand the immediate danger of using SDNEnhancements.

3.1 Experiment Setup

We begin by describing the workloads and topologies used in our study. We conduct our study in Mininet [32] (an emulator) using a $k = 4$ Fat-Tree data center topology [3]. We investigate the SDNApps and SDNEnhancements under both realistic [7] and synthetic workloads (described in [3]). We performed our tests on a 2.80GHz quad core Intel Xeon PC with 16GB of memory running Ubuntu 14.04.

SDNEnhancements. We studied two different and representative SDNEnhancements:

- **TCAMOptimizer:** an SDNEnhancement that aims to maximize TCAM utilization. This SDNEnhancement is modeled after the optimizations discussed in [25].
- **ConflictResolver:** a canonical conflict resolving and resource management SDNEnhancement modeled after Statesman [48].

3.2 Implications of SDNEnhancements

In our study, we compare the aggregate network bandwidth under several different scenarios: *None*, no traffic engineering (provides us with a lower bound on performance); *Hedera*, the traffic-engineering SDNApp is used with no SDNEnhancements (provides us with an upper-bound on performance); *TCAMOptimizer*, Hedera is run with the TCAMOptimizer; *ConflictResolver*, Hedera is run with the ConflictResolver; *ALL*, Hedera is run with both SDNEnhancements.

SDNApp Efficiency: In Figure 1, we compare the aggregate network bandwidth against the number of TCAM entries used by Hedera. Recall, the goal of the SDNApp is to maximize network bandwidth utilization while the goal of the TCAMOptimizer is to minimize memory utilization. We observe that applying TCAMOptimizer reduces TCAM utilization by 57.5% but at the cost of performance (24.8% reduction in aggregate bandwidth). This reduction in bandwidth occurs because TCAMOptimizer substitutes fine-grained rules for coarse-grained rules which prevents Hedera from identifying some elephant flows. Similarly, we observe a decrease in aggregate bandwidth when ConflictResolver is used because Hedera's reaction latency increases thus prolonging periods of congestion and reducing bandwidth for congested flows.

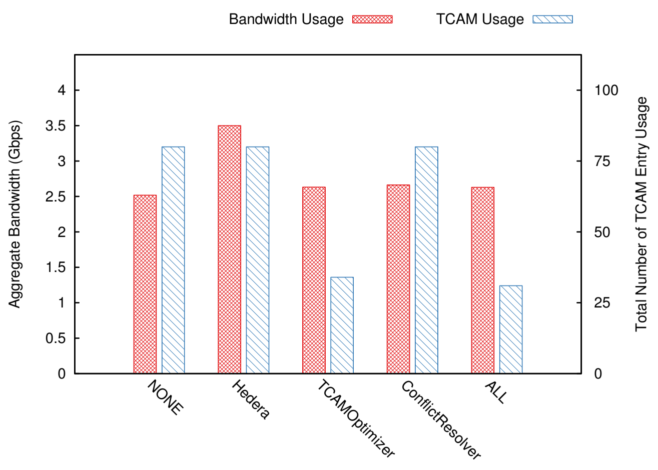


Figure 1: Aggregate Bandwidth and TCAM Usage.

4 RETHINKING CONTROLLER ARCHITECTURES

The last two sections highlight several alarming problems: first, modern controllers lack appropriate primitives to support SDNApps, and second, adhoc integration of SDNEnhancements, which provide these missing primitives, can result in catastrophic consequences. Existing design choices for attacking these problems broadly fall into three categories.

First, introducing new abstractions that empower SDNApps and SDNEnhancements to detect and react to each other (e.g., Athens [5]). This approach is prone to oscillations and convergence issues [5]. Furthermore, it unnecessarily burdens SDNApp developers to write code for conflict detection and resolution. Second, developing new controllers that allow SDNApps and SDNEnhancements to directly specify their internal constraints and objectives; the controller then solves an optimization problem to automatically arrive at an optimal solution (e.g., extending SoL [18] to support composition). This approach requires SDNApp developers to agree on a common meta-objective on which the controllers can optimize and to transform their internal objectives into this meta-objective. Finally, forcing developers to write monolithic SDNApp that include SDNEnhancements, e.g., Niagara [24] which combines TE with TCAM optimizations. Unfortunately, this does not scale and increases the barrier for developing new SDNApps or SDNEnhancements. These three alternatives all place unnecessary burdens on the SDNApp developers countering one of the motivating factors of SDNs: *ease of developing custom SDNApps*.

Instead, we take inspiration from the compiler community and argue that SDN controllers, SDNEnhancements, and SDNApps should be redesigned to mirror the interactions between compilers, compiler optimizations, and developers. Specifically, the compiler subsumes and controls all optimizers and uses a set of compiler-flags to determine the set of optimizations to perform and how to perform them: the flags are, in turn, controlled by the developer. For example, developers can specify "-01" to turn off all optimizations and improve compilation speed, or specify "-fno-elide-constructors" to turn off a specific optimization. Similarly, the controller should subsume and control, rather than be disjointed from, the SDNEnhancements and the controller should leverage SDN-Flags from the SDNApps to determine how to apply the SDNEnhancements to the SDNApps.

Our compiler-inspired approach explores a point in the spectrum of available design choices, alternatively we could raise the level of abstraction, by introducing a higher-level language [37, 44, 49] for programming SDNApps – this interface shifts the burden from the developer to the runtime which automatically infers the set of transformations that are allowable. Unfortunately, higher-level programming APIs have received little adoption from the industry due to the overheads required to train developers to learn the new language. Motivated by our desire to integrate into currently deployed controllers, e.g., ONOS, Floodlight, FAUCET and OpenDaylight, we choose the former approach of enriching the current abstractions and, thus, we apply a paradigm intimately that the developers are familiar with – compiler optimizations.

4.1 Compilers for SDNs

Next, we show how interactions within the SDN ecosystem can be represented within a compiler-style abstraction. We focus on the SDN control messages, on policies and SDNEnhancements.

At a high level, a traditional compiler takes in source code, transforms it into an intermediate representation (a more general instruction set). In the intermediate form, code is grouped into blocks and a DAG is created capturing the control flow between blocks. The compiler applies a set of local and global optimizations (transformations) to the resulting DAG. The local optimizations focus on a block of code, whereas global optimizations operate across blocks of code.

Next, we show how we map concepts within the SDN ecosystem into the traditional compiler scenarios. We focus on (1) the individual control messages that make up the SDN assembly code, (2) a novel abstraction for capturing logical blocks of messages, (3) a method for inferring control flow (and dependencies) between blocks, and (4) a novel set of SDN-Flags.

SDN Instruction Set: In SDN, the controller configures the network using a set of low-level control messages discussed earlier (Section 2) – OpenFlow uses rules (a pair of match and action tuples). These are akin to low level assembly code. SDNEnhancements transform these control messages into control messages, e.g., local SDNEnhancements transform messages by changing the match or action attributes and global SDNEnhancements transform messages by changing their temporal ordering or spatial location in the network.

Transactional Policy: Unlike compilers which translate high-level source to low-level assembly, the controller accepts low-level commands from SDNApps and directly installs them into the network. These low-level commands have forced SDNEnhancements to generate different meta-abstractions for capturing higher-level intent on which to perform optimizations, e.g., "proposed state" by Statesman [48] or "Transactions" by STN [10] and ESPRES [40].

To address this lack of abstractions, we define a uniform abstraction on which all SDNEnhancements can operate. To do this, we select the lowest common denominator: a network path.

More formally, a transactional policy, $t_{x_i,y_i} = \{m_1^i, m_2^i, ...\}$, is akin to a "code block" and is a group of SDN instructions required to configure a network policy between two hosts x_i and y_i (or groups of hosts)[1].

Thus, we formalize interactions between an SDNApp and the network (and, in turn, the SDNEnhancements) as a policy set, T, where T is:

$$T = \{t_{x_1,y_1}, t_{x_2,y_2}, ...\}$$

[1]This path level abstraction echoes recent efforts in SDNs to build optimization-based and monitoring-focused frameworks predicated on network paths.

Given this definition, an SDNEnhancement is a function, E, that transforms one transactional policy, $t_{x,y}$, into an "optimized" transaction policy $t'_{x,y}$:

$$t'_{x,y} = E(t_{x,y})$$

With these definitions in mind, we can also formalize situations where SDN-Flags are required by analyzing the interactions between policies and packets in the data-plane. Specifically, we can examine a set of packets:

$$P = \{p_1, p_2, ...\}$$

where each packet, p_i, represents traffic between x_i and y_i that will be processed by policy t_{x_i,y_i}. By applying the transactional policies T, a packet p_i would gain a set of decisions $d_i = T(p_i)$, including the routing path, dropping decision, queuing time, e.t.c. We compare the decisions before and after applying the SDNEnhancement function E:

$$T(P) = \{d_1, d_2, ...\}$$

$$(E \circ T)(P) = \{d'_1, d'_2, ...\}$$

and

$$N = |\{i | d_i \neq d'_i\}|$$

When there is a difference in behavior, then there is potentially a need for SDN-Flags. Depending on the sources of and the cause of these behavioral differences, the developers can employ different SDN-Flags to eliminate or minimize the differences. In Section 4.2, we characterize these SDN-Flags and discuss how developers can introduce them.

Transactional Dependencies & Intermediate Representation: This paper does not explicitly tackle conflicts between SDNEnhancements or verification of SDNEnhancements. Instead, we present a high-level description of ongoing efforts to do this. Conflict detection and verification requires an intermediate representation that abstracts syntactic details and a notion of dependencies that formalizes conflicts.

We infer dependencies between transaction by building on the definitions provided in SDNRacer [34] and LegoSDN [12]. For intermediate representation, we use Header Space Analysis which captures the reachability policies and augments it to include QoS-based policies. Coupled with dependencies, the intermediate representation enables us to reason about conflicts between SDNEnhancements and verify policies.

4.2 Modeling Optimization Flags

SDN-Flags, like compiler flags, are designed to allow developers (and consequently the SDNApps) to limit the class of transformations that can be applied rather than the set of SDNEnhancements: the SDN-Flags (flags) do not specify specific SDNEnhancements (optimizations) only transformations. This level of indirection frees the SDNApp developer from having to understand the SDNEnhancements that will be run in the network.

In modeling SDN-Flags, we aim to support a large variety of operational networks. Thus, we study the OpenFlow specification to understand the space of potential transformations that can be performed, independent of any specific SDNEnhancements. In Table 2, we present an exhaustive list of these transformations and a representative list of SDNEnhancements that employ them (when available). Transformations can be classified along four dimensions:

modifications to the rule's match field (e.g., merging, duplicating, or splitting rules); modifications to the rule's actions (e.g., changing ports); modifications to the rule's temporal property (e.g., reordering or delaying rules); and modifications to the rule's spatial properties (e.g., changing the switch that a rule is installed in).

Dimension of Transformation	Type of Transformation	Example SDN-Enhancement	SDN Flags
Match Fields	Merges Rules	[46]	{IO}
	Splits/Duplicates Rules	[45]	
Action List	Adds Actions	None	{AD}
	Reorders Actions	None	
	Deletes Actions	None	
Spatial (Location)	Changes Destination Switch to Install Rules	[25, 51]	{LS}
Temporal (Ordering)	Re-Orders Rules	[40]	{PF}
	Delays Rules	[14, 48]	
NULL	Deletes SDN Message(s)		

Table 2: List of Potential Transformations Made by SDNEnhancements and the SDN-Flags Specified by SDNApps.

Controlling SDNEnhancements with SDN-Flags: In Table 2 (column 4), we present SDN-Flags that SDNApps can use to control transformations that violate correctness or efficiency. We note that the current SDN-Flags are both general and simultaneously specific because the control interface between the control plane and the forwarding tables in both OpenFlow and P4 switches are limited and narrow. Thus, the set of potential transformations that any SDNEnhancements may perform is finite and extremely limited. Next, we elaborate on how these SDN-Flags can be used to address the issues presented in Section 2:

- *Input-Output dependence {IO}:* specifies that the SDNApp's inputs are a function of the rules installed in the network (the SDNApp's output). This SDN-Flag allows the controller to ensure the correctness of the SDNApps by circumventing SDNEnhancements whose transformations lead to coarser granularity rules. For SDNEnhancements whose transformations result in other or no transformations, the controller simply ensures that information for the finer-granularity rules are coalesced (nothing is done or required for rules which result in equivalent granularity). For example, if a controller applies a TCAM-Optimizer SDNEnhancement that merges rules into coarser granularity rules to the TE-App, which has an Input-Output dependence, then Mozart may bypass the SDNEnhancement for such SDNApps (or perform some other operation) which would preserve the Input-Output dependence.
- *Action-Dependence {AD}:* specifies that the SDNApp's functionality and correctness are tied to the actions created and inserted into the FlowMods.
- *Location-Specific {LS}:* specifies that the SDNApp's functionality and correctness are tied to the specific switches selected for the path.
- *Push-Flag {PF}:* When reacting to a failed link or an intruder, it is imperative to react first and to optimize second. For these use-cases, we provide SDNApps with a Push-Flag that signifies urgency. This SDN-Flag allows the controller to directly perform the SDNApp's proposed changes into the network while simultaneously applying the SDNEnhancements to these actions. When the SDNEnhancement returns the optimized (transformed) rules, the controller replaces the SDNApp rules with the optimized version.

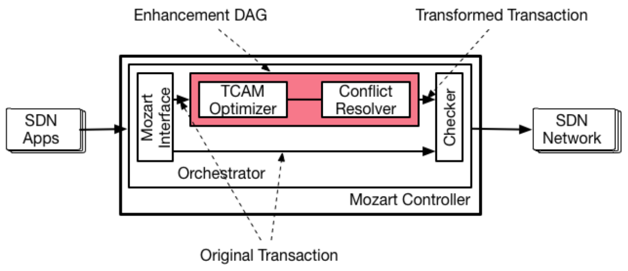


Figure 2: Re-Designed SDN Controller.

Takeaways. SDNApps encapsulate a rather simple control loop with a limited number of variations (Section 2). Through an examination of the specification, we observe that the space of transformations is limited (Table 2). The implication of these insights is that a limited set of SDN-Flags will cover a dominant number of SDNApps. Additionally, this constrained transformation space and our formalizations provide the groundwork for a system that automatically generates SDN-Flag for SDNApps – a system we plan to explore in the future.

5 MOZART

In Figure 2, we present Mozart a redesign of the modern controller architecture that applies compiler-optimizations philosophies to SDNEnhancements. Mozart exposes a novel interface to the SDNApps which enables these SDNApps to bundle SDN commands into *transactional policies* (Section 4.1) and to annotate the transactions with SDN-Flags (Section 4.2). The controller includes an Orchestrator, similar to compiler tools, that orchestrates SDNEnhancements, applies them to SDNApps, and ensures that SDN-Flags are respected. In Mozart, SDNEnhancements are integrated into the controller as isolated modules within the Orchestrator and communication between them is through function calls.

Interfaces: Mozart defines well-specified interfaces for how SDNApps should interact with the controller and for smoothly integrating the SDNEnhancements into the Orchestrator.

The SDNApp interface, Figure 3, specifies a call that Mozart exposes to all SDNApps: `apply()`. Using `apply()`, an SDNApp can specify a `Transaction`, i.e., a bundle of SDN instructions, to apply to the network rather than individual instructions (or messages). Furthermore, SDNApps may annotate transactions with SDN-Flags either one SDN-Flag for the entire transaction or an SDN-Flag for each instruction in the transaction.

The SDNEnhancement-interface, Figure 4, enables the Orchestrator to manage SDNEnhancements and promotes interoperability between SDNEnhancements. To this end, the interfaces specify the set of functions that each SDNEnhancement must implement.

```
public interface Mozart {
 Class Transaction{
   Map <SDNMessage, SDNHint> bundle;
    List <SDNHint> global;
 }

 public void apply(List <Transaction>);
}
```

Figure 3: Interface Exposed to SDNApps by Mozart.

Each SDNEnhancement must implement the following functions: `init()`, `process_transaction()`, and `configure()`. `process_transaction()` takes a list of transactions as input and optionally returns a list of (zero or more) transactions.

```
public interface Enhancement {
   public List<Transaction> process_transaction
   (List <Transaction>);
   public void init();
   public void configure (Map <String, String>);
}
```

Figure 4: Interface for SDNEnhancements.

When the Orchestrator initializes a new SDNEnhancement, due to a new DAG or modifications to an existing DAG, it calls the SDNEnhancement's `init()` function. As network administrators modify configurations for an SDNEnhancement, the Orchestrator calls `configure()` to reconfigure the SDNEnhancement. When an SDNApp calls `apply()`, the Orchestrator accepts the transaction and passes it through the set of SDNEnhancements listed in the DAG: then `process_transaction()` is called for each SDNEnhancement – the output of one `process_transaction()` is used as input for the next `process_transaction()`.

Orchestrator: Runs within the controller and accepts an administrator-defined configuration: a linear DAG of SDNEnhancements to apply to each SDNApp. The Orchestrator accepts a transaction from an SDNApp, through the `apply()`, determines the DAG for the SDNApp, and propagates the transaction through SDNEnhancements in the DAG. The output of the final SDNEnhancement (in the DAG) is fed to the Checker which compares the transformed transactions against the original transactions to ensure that the transformations are valid with respect to the specified SDN-Flags.

At a high level, the Checker verifies that for each SDN-Flag specified none of the violating transformations (in Table 2) are applied to the transaction. For example, when the {IO} SDN-Flag is specified, the Checker verifies that "merge rule" transformations are not applied – if applied, the Checker reverts the transaction to the original transaction. When the {PF} SDN-Flag is specified, the Orchestrator monitors the chain of SDNEnhancements and if they take longer than a predefined timeout, δ, to process the transaction, then the Orchestrator directly applies the original transaction to the network and subsequently updates the network with the optimized (transformed) transaction after the SDNEnhancements are done.

5.1 Using Mozart

In Mozart, the network operator specifies a linear DAG of SDNEnhancements to apply to each SDNApp – the Orchestrator uses this DAG to determine orchestration. The operators also specify a list of SDNEnhancements that cannot be avoided, e.g., *a security SDNEnhancement should have priority over SDN-Flags specified by any SDNApps*.

The developer writes SDNApps to leverage the interface and employs SDN-Flags when necessary. There are several options for the developer:

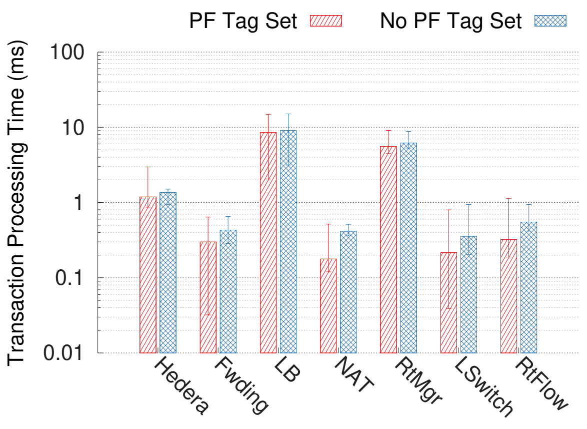
- **Fine Granularity Use of SDN-Flags:** Either rewrite the SDNApp to integrate SDN-Flags at a fine granularity, e.g., an SDN-Flag for each transaction, similar to how pragmas and annotations are included in programs to aid optimizers.
- **Coarse Granularity Use of SDN-Flags:** Or, specify the SDN-Flags at a coarser granularity, e.g., specific SDN-Flags for edge devices and different SDN-Flags for core devices. This direction eliminates the burden of rewriting the SDNApp while providing the developer with the ability to benefit from our system. These SDN-Flags can be specified either through command line arguments (or in a configuration file). More concretely, the SDNApp developer can specify the set of SDN-Flags to apply to different function calls, e.g., for edge devices versus for core devices.
- **Automated SDN-Flag Generation:** We could develop a simulation framework that enables Mozart to automatically learn the appropriate SDN-Flags based on operators specified invariants on packets and data-plane behavior (e.g., SDNEnhancements should not impact performance by more than $X\%$, or SDNEnhancements should not consume more than $Y\%$ of the network's resources). Given these invariants, Mozart can use a simulator to compare the performance of SDNApps with and without SDNEnhancements and explore the different SDN-Flags using a greedy heuristic (e.g., Simulated Annealing) to effectively discover the appropriate SDN-Flags.

Employing SDN-Flags requires administrators to explore a trade-off between invasiveness and resolution; the finer the granularity, the more involved the changes are to existing SDNApps. Whereas with more automated insertion of SDN-Flags, naturally the administrators lose control over precise SDNApp behavior. We show in Section 7, there are significant benefits when SDN-Flags are applied at a coarse granularity.

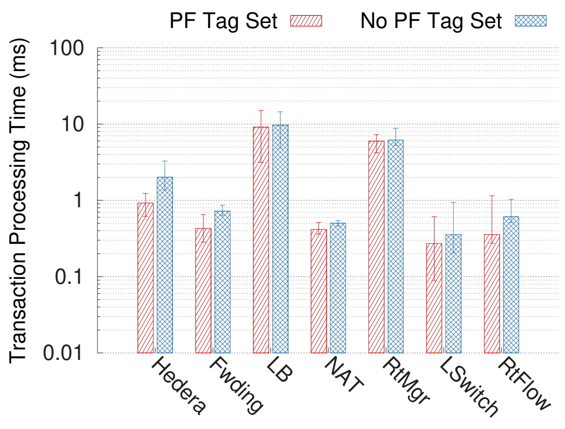
6 PROTOTYPE

We developed prototypes of Mozart by integrating our designs into two production quality controllers – one used at Google (i.e., FAUCET [6] a fork of RYU). Our SDNApps mirror crucial properties of production SDNApps, e.g., Hedera's control loop is philosophically similar to Microsoft's-SWAN [20] and Google's-B4 [22] both of which feed switch statistics into the traffic-engineering algorithm.

Mozart's design differs from a traditional controller in two ways: it exposes an interface for applications to utilize our primitives and it explicitly incorporates SDNEnhancements functionality. We



(a)



(b)

Figure 5: (a) {PF} SDN-Flags' Impact on Transaction Processing Time. (b) {PF} SDN-Flags' Impact with Multiple Simultaneous SDNApps.

Class of Code	Modified Instances	LoC
SDNEnhancements	ConflictResolvers	134 (20%)
	TCAMOptimizer	119 (11.4%)
SDNApps	Hedera	18 (0.4%)
	Forwarding	33 (1.7%)
	Load Balancer	13 (0.4%)
	NAT	18 (1.5%)
	Route Manager	19 (1.1%)
	Five versions of Learning Switch	18 (1.2%)
	Route Flow	13 (0.3%)
Controller	Floodlight	1326 (1.5%)
	Ryu	116 (0.6%)

Table 3: Lines of Code Changed.

chose to explicitly incorporate SDNEnhancement functionality as a module as this allows us to explicitly inform an SDNEnhancement of the primitives used by each SDNApp. Moreover, we modified the controller to monitor and log the transformations made by the SDNEnhancements for debugging purposes. Our prototypes are built atop the Floodlight controller in 1326 Lines of Code (LoC) and Ryu controller in 116 LoC. Mozart interacts with the SDNEnhancements using functions calls. The SDNEnhancements and the SDNApps have been modified to generate SDN-Flags and to use SDN-Flags respectively.

Changes to SDNEnhancements: We changed the TCAMOptimizer, 119 LoC (11.4%), and the ConflictResolver, 134 LoC (20%), to provide the functionality discussed in Section 5. Our modifications to the SDNEnhancement, the SDNApp, the Floodlight controller and the Ryu controller are detailed in Table 3.

Changes to SDNApps: We changed seven SDNApps to leverage our SDN-Flags and Mozart's interface. From Table 3, we observe that the changes to the SDNApps were minimally invasive (generally less than 2% of the codebase was modified). Note for Ryu, we had five versions of the Learning Switch SDNApp, and we modified all five versions.

7 EVALUATION

To understand Mozart's effectiveness in maintaining application performance in the face of SDNEnhancement transformations, we evaluate Mozart against the SDNEnhancements and SDNApps discussed in Section 6. We investigate Mozart under a combination of synthetic and realistic traces [7] and with a variety of data center topologies. This diversity allows us to draw general conclusions about our abstractions and their implications. In evaluating Mozart, we aim to answer the following questions:

- Is Mozart able to effectively improve an SDNApp's performance? (§ 7.2)
- What fraction of Mozart's benefits are achieved when Mozart is applied in a backward compatible manner (requiring no code changes to the SDNApps)? (§ 7.3)
- How much overhead does Mozart introduce? (§ 7.4)
- How much additional work does Mozart's interface introduce when SDNApps are updated? (§ 7.5)

7.1 Experiment Setup

We begin by describing the workloads, and the topologies used in our evaluations. We conduct our experiments in an emulator, Mininet [32], and with a simulator. The emulator allows us to understand the accuracy and efficacy of Mozart whereas the simulator allows us to understand the scaling implications of Mozart. In both our emulations and simulations, we consider the SDNApps and SDNEnhancements discussed in Section 6. We consider a Fat-Tree topology [3] and investigate both realistic and synthetic workloads. For realistic workloads, we consider the traffic patterns for a medium data center [7]. For the synthetic workloads, we consider the best case (Random) and the worst case (Stride) traffic matrices used in recent data center proposal [3, 4]. The stride pattern has multiple flows from the same source edge switch to the same destination edge switch.

Simulator: In the absence of a large-scale testbed to study the overheads and scaling implications of Mozart, we instead developed a simulator to model the network. We simulate various network events and the corresponding messages exchanged between the network devices and the controller (e.g., the control messages sent to the controller by the switches when statistics are requested or when the switch is powered on/off). By simulating only network events, our simulator is transparent to the SDNApps, the SDNEnhancements, and Mozart. This transparency allows them to operate as usual ensuring that we can objectively evaluate the overheads of Mozart. This approach allows us to focus on the performance of Mozart in a large-scale setting while being unconstrained by the size and topology of our local resources. In our simulations, the network controller is deployed on a 2.80GHz quad core Intel Xeon PC with 16GB of memory running Ubuntu 14.04.

Unless explicitly specified, our default experiments are run on the Fat-Tree topology, with 20 nodes, 16 hosts, 1Gbps links and with the stride traffic pattern.

7.2 Implications of Mozart

We begin this section, by investigating the high-level impact of Mozart on the broad set of SDNApps evaluated then we focus on two specific SDNApps to understand SDNApp-specific performance: in particular, to illustrate the interactions between Mozart and the two classes of SDNApps, we focus on a proactive SDNApp and a reactive SDNApp.

Broad Analysis: In Figures 5 and 6, we analyze the impact of two specific flags on our SDNApps. In general, the flags have varying benefits which are correlated to the functionality of the different SDNApps.

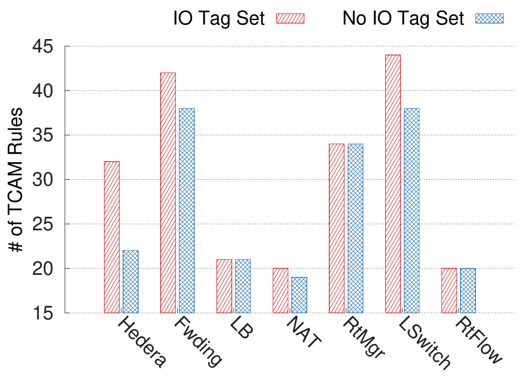


Figure 6: Number of TCAM Entries when {IO} is Enabled.

To better understand the impact of Mozart, we examined the average transaction processing time when {PF} is enabled. Figure 5 shows the transaction processing time with and without {PF} enabled for several SDNApps. We observe that {PF} does, in fact, decrease processing time demonstrating the benefit of introducing and using such a flag. Next, in Figure 6 we observe the impact of the {IO} Flag on the number of TCAM rules in the network. We observe that the flag does inflate the number of rules however this inflation is modest and acceptable in light of the potential benefit: namely, improved performance.

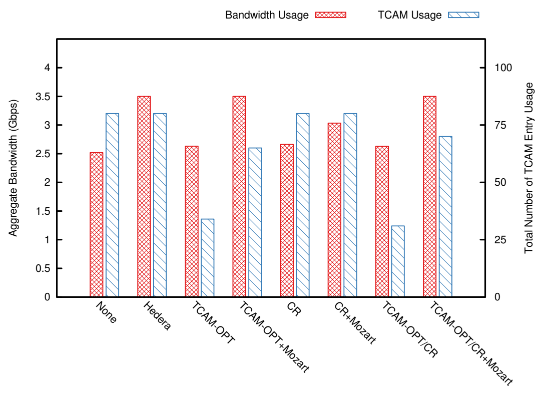


Figure 7: Aggregate Bandwidth and TCAM Usage.

Proactive App (Hedera): Next, we drill into a proactive SDNApp and compare the aggregate network bandwidth under several different scenarios: *None* scenario, no traffic engineering provides us a lower bound on performance; *Hedera* scenario, Hedera traffic-engineering is used with no SDNEnhancements – this provides us with an upper-bound on performance; *TCAM-OPT*, Hedera is run with the TCAMOptimizer; *CR*, Hedera is run with the ConflictResolver; *TCAM-OPT/CR*, Hedera is run with both the ConflictResolver/TCAMOptimizer; *TCAM-OPT+Mozart*, Hedera is applied with the TCAMOptimizer and Mozart; *CR+Mozart*, Hedera is applied with the ConflictResolver and Mozart; *TCAM-OPT/CR+Mozart*, Hedera is applied with both ConflictResolver and TCAMOptimizer and Mozart.

In Figure 7, we compare the aggregate network bandwidth against the number of TCAM entries used by Hedera. We observe that applying the TCAMOptimizer, reduces TCAM utilization by 57.5% but at the cost of performance (24.8% reduction in aggregate bandwidth). This decrease occurs because the TCAMOptimizer eliminates Hedera's ability to effectively determine which flows are elephants. Similarly, we observe a decrease in aggregate bandwidth when ConflictResolver is used because Hedera's reaction time is increased thus prolonging periods of congestion and reducing bandwidth for congested flows.

In applying Mozart, we observe that bandwidth is improved to within the optimal solution. While Mozart drastically improves Hedera's performance, we observe that the efficiency of the TCAMOptimizer is reduced – the TCAMOptimizer is only able to achieve 18.2% of TCAM usage saving (the fourth bar). This performance to TCAMOptimizer trade-off occurs because Mozart improves performance by limiting coalescing on certain OpenFlow entries. The improvement over the ConflictResolver on the other hand occurs because Mozart temporarily ignores ConflictResolver and retroactively applies the optimization of the impact of the SDNEnhancements.

Reactive App (RtFlow): Lastly, we evaluate the impact of the SDNEnhancements on a reactive SDNApp, we focus on the route-setup. In this scenario, the TCAMOptimizer has no impact and thus we exclude it and focus solely on these two scenarios: *CR* and

CR+Mozart. We observe that ConflictResolver has a similar impact in that it reduces the ability of the SDNApp to install paths and to react to the injected failure events. In Figure 8, we present a time series of the number of active flows within the network. We observe that with *CR* during the initial 3.3 seconds there are no active flows and that at the second 50 there is another dip in the number of active flows when a link is deleted from the network. Unlike, *CR*, we observe that *CR+Mozart* has a much lower initial ramp of phase and time to recovery with *CR+Mozart* being 7.8 times and 44.8 times faster than *CR*. This displayed the benefits of employing Mozart[2].

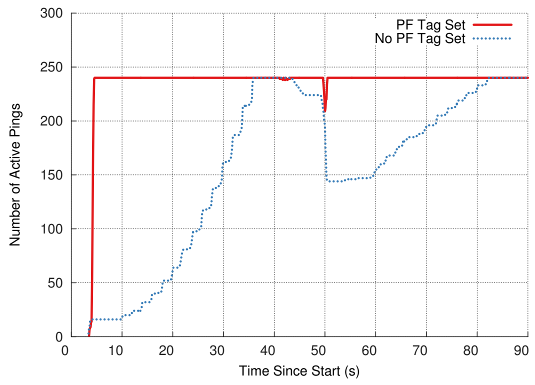


Figure 8: Ping Latency in Link Failure Experiment.

7.3 Resolution of Mozart

Fundamentally, Mozart introduces a set of abstractions that facilitate exchange of information. As discussed in Section 4, these interfaces can be used at a varying-resolutions:

- *static*, with the same SDN-Flags applied to all OpenFlow-messages or at a fine-resolution,
- *dynamic*, the default behavior, with SDN-Flags judiciously applied to each OpenFlow-message.
- *static-dev*, to support ease of integration, in Section 4, we suggested that SDN-Flags be applied statically at the granularity of function calls and device types. More concretely, the SDNApp developer can specify the set of SDN-Flags to apply for different function calls, for edge devices, and for core devices. The decision to delineate device along the core-edge boundaries builds on recent trends to separate the core from the edge [11, 30, 43].

There is a trade-off between invasiveness and resolution; the finer the granularity, the more involved the changes are to existing SDNApps. At one extreme, *static* requires absolutely no change and at the other extreme *dynamic* requires changes to the SDNApp; however, these changes are minimal. As a middle-ground option, *static-dev*, requires a simple addition – the inclusion of a configuration file.

[2]We note that while our implementation of ConflictResolver takes about 10 seconds to process, the relative speeds are subject to change given different implementations of ConflictResolver. Furthermore, while ConflictResolver potentially improves network performance it can result in transient periods of conflicting resource allocations.

We observed that with *static-dev* the simple distinction between core and edge is sufficient to maximize the trade-off between SDNApp accuracy and SDNEnhancement efficiency. *static-dev*'s performance and efficiency are close to *dynamic* without incurring the overheads of re-writing the SDNApp. Intuitively, *static-dev* performs close to *dynamic* because the information required to detect congestion is present at the edge, and, additionally, most of the congestion occurs within the edge. This result demonstrates the feasibility of adopting Mozart without invasive modifications to the applications. More generally, we believe that *static-dev* is broadly applicable to other SDNApps because, in most SDNApps, there is a distinction in the functionality applied at the core from that applied at the edge of the network.

Orthogonally, with *static*, we observed that blindly applying the same SDN-Flags impacts and hurts performance. Fortunately, we believe that *static-dev* provides a promising and non-invasive step forward for a broad set of SDNApps.

7.4 MicroBenchmarks

We examine the overhead of employing Mozart and investigate how these overheads scale along two dimensions. First, in terms of additional latency for the Orchestrator to compose services and evaluate the SDN-Flags. Second, in terms of the throughput of the controller. To do this, we evaluate Mozart using our simulator.

We examine the throughput and latency for processing OpenFlow-messages on a number of topologies with varying sizes. In Figure 9, we focus on the largest data center topology evaluated: Fat-tree with 2000 hosts and 500 network devices. From Figure 9, we make two observations: first, that the overheads imposed by Mozart are sub-linear and second, the overheads imposed are minimal and acceptable with additional SDNEnhancements imposing a 1.58% overhead to latency and no observable overhead to throughput.

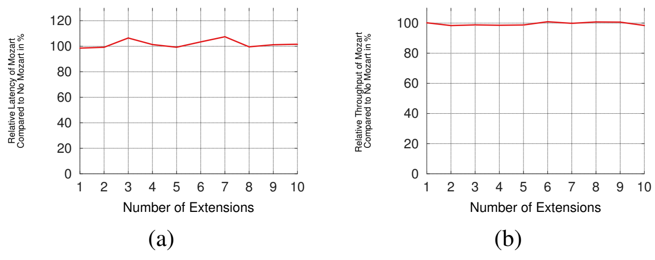


Figure 9: (a) Relative Latency of Mozart Compared to No Mozart in %. (b) Relative Throughput of Mozart Compared to No Mozart in %.

7.5 Implication of SDN-Flags on SDNApp Evolution

Finally, we conclude by examining the impact of Mozart on a developer's ability to manage an evolving codebase. Here we focus on a specific SDNApp on Ryu (Learning Switch). Currently, Ryu comes with five versions of this SDNApp – one for each of the different versions of the OpenFlow interface[3]. In Figure 10, we plot the number of transactions required. We observe that the only change happens between version 1.2 and 1.3 when an additional transaction is added due to new feature in the specification (i.e., Table Miss).

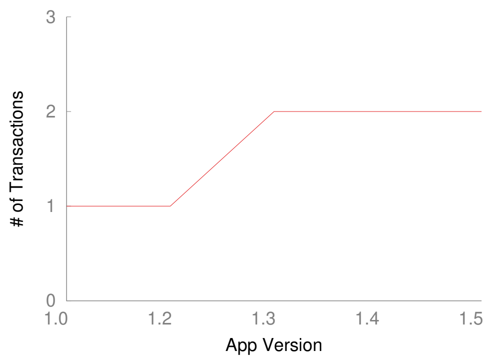


Figure 10: Number of Transactions for Different Versions of the Learning Switch SDNApp.

8 DISCUSSION

Implications on security and other properties? We explored the implications of SDNEnhancements on efficiency, complexity, and fidelity. There are other dimensions along which SDNEnhancements may impact an SDNApp. For example, security, network utilization, performance, isolation, etc. As part of future work, we intend to explore these dimensions. For example, we could integrate Mozart with Rosemary [47], a security oriented controller architecture, to analyze how SDNEnhancements impact security policies. Moreover, we plan to more concretely explore the connection between SDNEnhancements and complexity in network management, by analyzing how SDNEnhancements impact the ability of network operators to debug networks using common tools [38, 39].

Does ignoring an SDNEnhancement obviate its benefits? The single biggest limitation of Mozart is that, in certain situations, Mozart acts in a binary fashion: a subset of SDN-Flags prevent transformations which may render an SDNEnhancement ineffective. Fortunately, we showed in Table 1 that multiple SDNEnhancements can provide the same property, e.g., consistent-updates [33, 40, 45], and one of these alternative SDNEnhancements may be able to preserve the SDNApp's correctness. As part of future work, we plan to design a more flexible Orchestrator that automatically replaces an SDNEnhancement rendered ineffective by SDN-Flags with an equivalent one with appropriate transformations.

Do our abstractions provide complete coverage? As the SDN-ecosystem becomes richer with more SDNApps and SDNEnhancements, our abstractions will naturally have to evolve. However, we note that since our abstractions are fundamentally tied to the core properties of FlowTable entries, we expect our abstractions will evolve at a significantly slower pace than that of the entire SDN-ecosystem.

[3]They support different versions of OpenFlow from 1.0 to 1.5 (Ryu does not offer built-in support for OpenFlow 1.1).

How do we handle interactions between multiple SDNApps? Multiple SDNApps can read and modify the same OpenFlow rules, this is akin to reading/writing to the same key-value stores. There are two ways to deal with this, both using SDNExtensions. Either the first writer wins approach taken by the Network-State management service or the capability/priority approach taken by Participatory Networking. Within our system, such conflicts can still exist and the SDNExtensions are responsible for tackling such conflicts. The existing conflict resolvers [14, 48] make two types of transformations: first, a null transformation which denies SDNApp and deletes the transaction. Second, a temporal transformation which delays transformations. Our system will not interfere with these resolvers.

How do we handle conflicts between SDNEnhancements? We explore the interactions between SDNEnhancements and SDNApps. Another more interesting set of interactions is that between a set of SDNEnhancements. SDNEnhancements are bound to conflict or to contradict each other. In this work, we do not explicitly address these issues, instead we take the first step towards addressing them by re-architecting controllers to explicitly include SDNEnhancements and explicitly compose SDNEnhancements. By making SDNEnhancements a more explicit member of the SDN ecosystem, conflicts can be readily detected, analyzed, and tackled. We plan to design a simulator to empirically detect these conflicts.

9 RELATED WORKS

Most notable work focus on re-architecting controllers to support scalability [31, 50], security [47], and reliability [12]. These works focus on improving the core architecture of the controller. Our approach is orthogonal and builds on them by proposing ways to extend the controller and directly incorporating SDNEnhancements. The most closely related work on SDN composition [14, 17, 23, 35, 37] focuses on providing SDNEnhancements that promote principled composition of SDNApps with different objectives [14, 35, 37] or SDNApps running on different controllers [23]. Our work presents a fundamental departure from existing work in the composition space, rather than focusing on the SDNApps, we concentrate on the SDNEnhancements. Thus, allowing us to introduce a similar level of rigor and understanding to SDNEnhancement-composition as we currently have for SDNApp-composition. Essentially, related work asks that each application should reimplement certain functionality, whereas Mozart extracts and pushes the functionality down to a lower and common layer: the controller. Furthermore, while related works focus on ensuring cooperation between SDNApps, we focus on ensuring cooperation between SDNApps and SDNEnhancements.

Additionally, unlike Mozart, Athens [5] and SOL [18] operate at the level of paths which places a key limitation on them: they can only detect harmful interactions between SDNApps and SDNEnhancements when the intersection of generated paths is empty. In Section 2, we showed that even if paths remain the same (a non-empty intersection), but other properties of rules are modified, e.g., by merging, then violations will exist. Mozart offers a fundamental advantage over them because it operates at a lower level of abstraction. SOL can not be easily modified to operate at this lower level because the lower level negates existing scaling optimizations forcing a fundamental redesign of its core algorithms. Athens can operate at this lower granularity; however, this change will significantly exacerbate Athens' existing scaling and complexity issue. Recall, Athens requires all SDNEnhancements and SDNApps to understand each other's logic and anticipate all interactions.

Our abstractions represent a natural extension of Operating System hints, such as X-tags [26], Intentional Networking [19] to the SDN's Network Operating System. Similarly, our SDN-Flags allow the SDNApps to expose their internal objectives in a qualitative manner without disclosing their internal structure. Unlike existing O.S. hints, our abstractions are motivated by domain-specific knowledge of design patterns and structure of SDNApps and SDNEnhancements. The design of our composition operators and configuration language are inspired by existing works on extensible system [21, 29, 52].

10 CONCLUSION AND FUTURE WORK

In this paper, we make the first attempt towards understanding and quantifying the implications of applying SDNEnhancements to SDNApps. We observe that SDN controllers are ill-equipped with poor primitives for supporting SDNApps and abstractions for enabling SDNEnhancements. Motivated by these observations, we argue for the design of a more powerful interface between the SDNApps and the SDN controllers – this interface allows for a systematic and principled inclusion of SDNEnhancements into the SDN ecosystem.

Our design and prototype implementation of Mozart is the first step towards a holistic controller architecture capable of supporting SDNEnhancements in a manner that does not compromise the simplicity promised by SDNs (or the performance, and efficiency of the SDNApps). We believe this idea of a holistic controller architecture capable of integrating and composing SDNEnhancements presents a rich field of future research and will become only more important as SDN deployments continue to grow. As part of future work, we aim to expand on our flags and tackle problems related to detecting conflicts between SDNEnhancements and verifying transformations made by an SDNEnhancement.

11 ACKNOWLEDGMENTS

We thank Eric Keller, Hyojoon Kim, and the anonymous reviewers for their invaluable comments. We also thank Chen Liang for his help with the prototype. This work is supported by NSF grants CNS-1409426 and CNS-1749785.

REFERENCES

[1] 2017. Ryu SDN Framework. https://osrg.github.io/ryu/.
[2] 2019. Project Floodlight. http://www.projectfloodlight.org/.
[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*.
[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*.
[5] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. 2014. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In *CoNext*.
[6] Josh Bailey and Stephen Stuart. 2016. FAUCET: Deploying SDN in the Enterprise. In *ACM Queue*.
[7] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*.
[8] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. 2011. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *SoCC*.
[9] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*.
[10] Marco Canini, Daniele De Cicco, Petr Kuznetsov, Dan Levin, Stefan Schmid, and Stefano Vissicchio. 2014. STN: A Robust and Distributed SDN Control Plane. In *ONS*.
[11] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. 2012. Fabric: A Retrospective on Evolving SDN. In *HotSDN*.
[12] Balakrishnan Chandrasekaran and Theophilus Benson. 2014. Tolerating SDN Application Failures with LegoSDN. In *HotNets*.
[13] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*.
[14] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*.
[15] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*.
[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*.
[17] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. 2018. Intent-Driven Composition of Resource-Management SDN Applications. In *CoNEXT*.
[18] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. 2016. Simplifying Software-Defined Network Optimization Using SOL. In *NSDI*.
[19] Brett D. Higgins, Azarias Reda, Timur Alperovich, Jason Flinn, T. J. Giuli, Brian Noble, and David Watson. 2010. Intentional Networking: Opportunistic Exploitation of Mobile Network Diversity. In *MobiCom*.
[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*.
[21] Norman C Hutchinson and Larry L Peterson. 1991. The X-Kernel: An Architecture for Implementing Network Protocols. *Software Engineering, IEEE Transactions on*.
[22] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with A Globally-Deployed Software Defined WAN. In *SIGCOMM*.
[23] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *NSDI*.
[24] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. 2015. Efficient Traffic Splitting on Commodity Switches. In *CoNEXT*.
[25] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the "One Big Switch" Abstraction in Software-Defined Networks. In *CoNEXT*.
[26] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. 2008. Network Exception Handlers: Host-network Control in Enterprise Networks. In *SIGCOMM*.
[27] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*.
[28] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*.
[29] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click Modular Router. In *ACM Transactions on Computer Systems (TOCS)*.
[30] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. 2014. Network Virtualization in Multi-Tenant Datacenters. In *NSDI*.
[31] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*.
[32] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in A Laptop: Rapid Prototyping for Software-Defined Networks. In *HotNets*.
[33] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*.
[34] Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin T. Vechev. 2015. SDNRacer: Detecting Concurrency Violations in Software-Defined Networks. In *SOSR*.
[35] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *HotNets*.
[36] Matthew Monaco, Oliver Michel, and Eric Keller. 2013. Applying Operating System Principles to SDN Controller Design. In *HotNets*.
[37] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software-Defined Networks. In *NSDI*.

[38] Tim Nelson, Da Yu, Yiming Li, Rodrigo Fonseca, and Shriram Krishnamurthi. [n.d.]. Simon: Scriptable Interactive Monitoring for SDNs *(SOSR '15)*.

[39] István Pelle, Tamás Lévai, Felicián Németh, and András Gulyás. [n.d.]. One Tool to Rule Them All: A Modular Troubleshooting Framework for SDN (and Other) Networks *(SOSR '15)*.

[40] Peter Perešíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. 2014. ESPRES: Easy Scheduling and Prioritization for SDN. In *ONS*.

[41] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *HotSDN*.

[42] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*.

[43] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. 2012. Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In *HotNets*.

[44] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. Fattire: Declarative Fault Tolerance for Software-Defined Networks. In *HotSDN*.

[45] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *SIGCOMM*.

[46] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. 2015. Too Many SDN Rules? Compress Them with MINNIE. In *GLOBECOM*.

[47] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. 2014. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *CCS*.

[48] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-state Management Service. In *SIGCOMM*.

[49] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*.

[50] Soheil Hassas Yeganeh and Yashar Ganjali. 2014. Beehive: Towards A Simple Abstraction for Scalable Software-Defined Networking. In *HotNets*.

[51] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM*.

[52] Erez Zadok and Jason Nieh. 2000. FiST: A Language for Stackable File Systems. In *USENIX ATC*.