A Call To Arms for Tackling the Unexpected Implications of SDN Controller Enhancements.

Theophilus Benson
Duke University

ABSTRACT

The last few years have seen a massive and organic transformation of the Software Defined Networking ecosystem with the development of enhancements, e.g., Statesman, ESPRES, PANE, and Athens, 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 or correctness.

In this paper, we present the first systematic study of the interactions between enhancements and SDN applications – we show that an application's performance can be significantly impacted by these enhancements: with the efficiency of a traffic engineering App reduced by 24.8%. Motivated by these insights, we argue for a redesign of the SDN controller centered around mitigating and reducing the impact of these enhancements. We demonstrate through an initial prototype and with experiments that our abstractions require minimal changes and can restore an SDN application's performance and efficiency.

CCS CONCEPTS

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

KEYWORDS

Software-defined Networking, Composition, Compilers

ACM Reference format:
Theophilus Benson. 2017. A Call To Arms for Tackling the Unexpected Implications of SDN Controller Enhancements.. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.

APNet'17, August 03-04, 2017, Hong Kong, China

ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3107006

https://doi.org/10.1145/3106989.3107006

1 INTRODUCTION

"More computing sins are committed in the name of efficiency (without necessarily achieving it) ."

—W.A. Wulf.

Software-defined Networking (SDN) aims to transform and simplify network management by exposing higher level APIs. With SDN, operators no longer configure networks through low-level commands but using higher level abstractions provided by SDN Applications (Apps). The push to deploy SDNs has exposed several underlying issues in the design of modern controllers, e.g., the controller's inability to perform congestion-free network updates [15].

A multitude of optimizations, which we call enhancements, have been developed to address these deficiencies and provide better composition of OpenFlow rules, better utilization of TCAM, consistent network updates, congestion free updates, etc. A representative list of these enhancements is provided in Table 1. Most enhancements are hidden from the controllers and Apps. They transparently intercept SDN control messages and perform optimizations before applying the messages to the switches.

The end result of this organic evolution of the SDN ecosystem is a disconnect between the App's view of the network and the actual network state: a disconnect between the control messages (forwarding rules) generated by an App and the forwarding rules stored in the data-plane which can impact an App's performance by as much as 24.8% (§ 3).

Interestingly, we observe that these enhancements are innocuously re-introducing complexity into the network by creating intricate dependencies and layers of indirections. In

Class of enhancement	Example	Description
Conflict-Resolvers	[5, 27]	Enforces resource allocation to different App
TCAM-Optimizers	[10, 16, 26, 30]	Minimizes switch memory (TCAM) utilization
Consistent updates	[15, 22, 25]	Updates network paths in a consistent manner
Invariant Checkers	[12, 13]	Checks to see if a network invariant holds, e.g., no cycles
App Composition	[3, 18, 20]	Combines rules from different Apps
Fault Tolerance Paths	[24]	Automatically creates backup paths to overcome link failure

Table 1: Taxonomy of enhancements.

fact, the indirection between the Apps and the data-plane is reminiscent of the indirection between the routing tables of traditional networking protocol and the hardware forwarding tables. Unfortunately, despite the growing presence and importance of these enhancements, there are few systematic (or holistic) studies of the implication of introducing enhancements.

Current work on SDN composition [3, 20] focuses on safely combining multiple Apps and tackling the complexity arising from sharing network resources. Instead, we focus on the enhancements applied to the resulting composed rules. For example, Pyretic [20] includes two composition operators for combining Apps' rules, the resulting combined rules are then optimized using enhancements, e.g., consistent update enhancements [15, 22, 25]. In this work, we focus on these enhancements.

In this paper, we take the first step towards understanding the impact of these enhancements on the SDN ecosystems and propose primitives and a framework, called Mozart, to help mitigate and stem this rising complexity. We borrow insights from the compiler community and their toolchains for orchestrating compiler optimizations. We propose a novel but simple interface that standardizes the interactions between the Apps, the controller, and the enhancements thus enabling us to systematically reason about the impact of enhancements. To mitigate the implications of enhancements on App, we propose a set of SDN-Flags, akin to compiler flags, that lets Apps specify the class of transformation that impact efficiency.

In summary, we make the following contributions:

- **Systematic Study of Complexity:** We present a systematic study of the implications of applying realistic enhancements to realistic Apps and show that an App's performance can be reduced by as much as 24.8% (§ 3).
- **SDN Abstractions:** We describe a set of interfaces and abstractions for mitigating and reducing the impact of these enhancements on Apps (§ 4).
- **Implementation & Evaluation:** We build a working prototype implementation of Mozart on Floodlight [23] and demonstrate the benefits of our primitives (§ 5).

2 MOTIVATION

In this section, we describe the fundamental structure of an App and present several of the simplifying assumptions that Apps make about the networks.

2.1 The Case for Enhancements

SDN Apps 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 enhancements by examining a canonical data center traffic engineering App, Hedera [2], and analyzing its interactions with the network. Hedera, Algorithm 1, aims to improve data center performance by detecting elephant flows (large 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 that load is balanced; and, (3) configuring new paths into the network with OpenFlow control messages.

```
1  while true do
       /* Get Network Input                                    */
2      foreach device in Network do
3          Counters.Append(device.GetStatistics())
4      end
       /* Control Function                                     */
5      Rules = BinPackingHeuristic(Counters)
       /* Send Output to Network                               */
6      foreach device in Network do
7          device.installRules(Rules)
8      end
9      Sleep100msecs
10 end
```

Algorithm 1: *Pseudocode for Hedera [2].*

In applying these control messages to the network, Apps, including Hedera, make the following assumptions about the network:

Infinite Hardware Resources: Apps assume an infinite amount of device memory (TCAM); However, TCAM space is limited in existing switches. Most can support 1K rules. The design choice of abstracting out hardware details and limitations is a common system design principles (e.g., OS provide Virtual Memory). However, unlike an operating system which provides adequate abstractions to support this, an SDN controller does not provide adequate abstractions. Thus to overcome this limitation, a class of enhancements [10, 16, 26, 30], *TCAM-Optimizers*, have been developed to provide the illusion of infinite memory.

Impact on Apps: These enhancements create optimized-rules that more efficiently utilize switch TCAM by merging, moving or splitting the rules generated by the App: essentially **transforming** an OpenFlow-message into *Coarser* Granularity or *Finer* Granularity messages. Unfortunately, certain Apps install rules of a certain granularity under the **assumption** that these rules can be used to collect the flow's metadata at the pre-specified granularity. The implication of 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 in Section 3).

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

Impact on Apps: In addition to modifying an OpenFlow-rule's match by making it coarser or finer, enhancements may also change the OpenFlow-rule's actions. In general, enhancements may transform an action in one of the following ways: (1) changing the network path by altering the interface associated with the action, (2) changing the reachability by changing the action, (3) changing QoS disciplines by changing the queues associated with the action. For Hedera, a direct implication of a path change (detour) is that the large flows, explicitly being load balanced, may be placed on identical links resulting in congestion. This path change may counter Hedera's control logic.

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

3 UNDERSTANDING ENHANCEMENTS

We now present empirical data to quantify the impact of enhancements on Apps: we focus on the App discussed in § 2 (Hedera) and analyze the reduction in aggregate bandwidth (*efficiency*) which allows us to understand the immediate danger of using enhancements.

Experiment Setup: We conduct our study in Mininet emulator using a k=4 Fat-Tree topology [1]. Note, while our initial study focuses on a small topology, we expect similar results under larger topologies. We investigate the App and enhancements under both realistic [4] and synthetic workloads (described in [1]). We performed our tests on a 2.80GHz quad-core Intel Xeon PC with 16GB of memory running Ubuntu 14.04.

Enhancements: We studied two different and representative enhancements:

- TCAMOptimizer: an enhancement that aims to maximize TCAM utilization by minimizing the number of TCAM entries used (modeled after [16, 26]).
- ConflictResolver: a canonical conflict resolving and resource management enhancement (modeled after [27]).

3.1 Implications of Enhancements

In our study, we compare the aggregate network bandwidth under the following scenarios: *None*, no traffic engineering (provides us with a lower bound on performance); *Hedera*, the traffic-engineering App is used with no enhancements (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 enhancements.

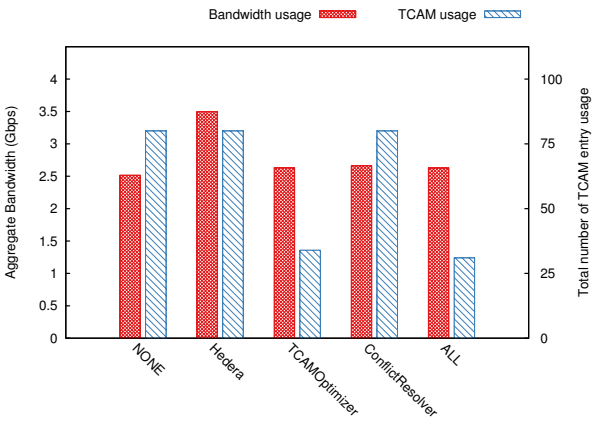

Figure 1: *Aggregated Bandwidth and TCAM usage.*

App Efficiency: In Figure 1, we compare the aggregate network bandwidth against the number of TCAM entries used by Hedera. Recall, the goal of the App is to maximize network bandwidth utilization while that of the TCAMOptimizer is to minimize TCAM usage. We observe that applying the TCAMOptimizer reduces TCAM usage 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 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.

4 RETHINKING CONTROLLER ARCHITECTURES

The last two sections highlight several interesting problems: first, modern controllers lack appropriate primitives to support Apps and second, ad hoc integration of enhancements, which provide these missing primitives, results in unexpected consequences. Existing design choices for addressing these problems fall into three categories.

First, introducing new abstractions that empower Apps and enhancements to detect and react to each other (e.g., Athens [3]). This approach is prone to oscillations and convergence issues [3]. Furthermore, it unnecessarily burdens App developers to write code for detection and resolution. Second, developing new controllers that allow Apps and enhancements to directly specify their internal constraints and objectives; the controller then solves an optimization problem to automatically arrive at an optimal solution [6]. This

Dimension of Transformation	Type of Transformation	Example enhancement
Match Fields	Merges rules	[16, 26]
	Splits/duplicates Rule	[25]
Action List	Adds actions	None
	Reorder actions	None
	deletes actions	None
Spatial (location)	Changes destination switch rule is installed on	[10, 30]
Temporal (ordering)	Re-orders rules	[22]
	Delays rules	[5, 27]
NULL	Deletes SDN Message	None

Table 2: *List of potential transformations.*

approach requires App developers to agree on a common meta-objective on which the controller can optimize and to transform their internal objectives into this meta-objective. Finally, enabling developers to write monolithic Apps that include enhancements, e.g., Niagara [9], which combines TE with TCAM optimizations; unfortunately, this does not scale and increases the barrier for developing new Apps or enhancements. These three alternatives place unnecessary burdens on App developers, countering one of the motivating factors behind SDN: the ease of developing custom Apps.

Instead, we take inspiration from the compiler community and argue that SDN controllers, enhancements, and Apps should be redesigned to mirror the interactions between compilers, compiler optimizations, and the developer. 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, a developer can specify "-01" to turn off all optimizations and improve compilation speed or "-fno-elide-constructors" to turn off a particular optimization. Similarly, the controller should subsume and control, rather than be disjoint from, the enhancements and the controller should leverage SDN-Flags specified by the Apps to determine how to apply the enhancements to the Apps.

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 [20, 24, 28] for programming Apps– this interface shifts the burden from the developer to the runtime which automatically infers the set of transformations that are allowable. Motivated by our desire to integrate into currently deployed controllers, e.g., Floodlight, ONOS and OpenDaylight, we choose the latter approach of enriching the current abstractions and applying a paradigm intimately familiar with today's developers – compiler optimizations.

4.1 Compilers for SDNs

At a high level, a traditional compiler takes in source code and transforms it into an intermediate representation (a more general instruction set). In this intermediate form, code is

```
public interface Mozart {
 Class Transactions{
   HashMap <SDNMessage, SDNHints> Bundle;
   ArrayList <SDNHints> Global;
 }

 public void Apply(ArrayList<Transactions>);
}
```

Figure 2: *Interface exposed to Apps by Mozart.*

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, whereas global optimizations operate across blocks.

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

SDN Instruction Set: An SDN controller configures the network using a set of low-level control messages. These are akin to low-level assembly code: enhancements transform properties and components of these control messages, e.g., local enhancements transform message by changing the match or action attributes, and global enhancements transform messages by changing their temporal ordering or spatial location in the network.

Transactional Policy: Unlike a compiler which translates high level source to low level assemble, the controller accepts low level commands from Apps and directly installs them into the network. enhancements, instead, work on groups or sets of commands.

To efficiently support enhancements, we define a uniform abstraction on which all enhancements can operate. To do this, we select the lowest common denominator: a network path. We define transactional policies: $T = \{t_{x,y}, t_{z,y} ... t\}$. A transactional policy, $t_{x,y} = \{m_1, m_2, m_3\}$, is akin to a "code block" and is a group of SDN instructions required to configure a network policy between two hosts x and y (or groups of hosts). [1] Given this definition, an enhancement is a function, F, that transforms one transactional policy, $t_{x,y}$, into an "optimized" transaction policy $t'_{x,y}$: $t'_{x,y} = F(t_{x,y})$

SDN Compiler Flags (SDN-Flags): Central to Mozart's design is a set of SDN-Flags that allows Mozart to understand and mitigate conflicts between Apps and enhancements. These SDN-Flags parallel traditional compiler flags. Recall, compiler flags enable the compiler to reason about how to apply optimizations. We expand on SDN-Flags in § 4.2

Transactional Dependencies & Intermediate Representation: This paper does not explicitly tackle conflicts

[1] This path level abstractions echoes recent efforts in SDNs to build optimization-based and monitoring-focused frameworks using paths

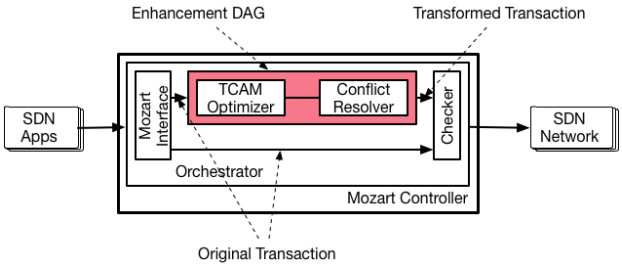

Figure 3: *Re-Designed SDN Controller.*

between enhancements or verification of enhancements. 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. To this end, we are working on extending existing work [12, 17] to infer dependencies and extract intermediate representations (header spaces). Coupled with dependencies, the intermediate representation will enable us to reason about conflicts between enhancements and verify policies.

4.2 Modeling Optimization Flags

SDN-Flags, like compiler flags, are designed to allow developers (and consequently the Apps) to limit the class of transformations, rather than enhancements, that can be applied to her App. This indirection frees the SDN App developer from having to understand the space of all possible enhancements that may be run in any network.

In modeling SDN-Flags, we aim to support a large variety of operational networks. Thus, we study the OpenFlow specification to understand, independent of any specific enhancements, the space of possible transformations that can be performed. In Table 2, we present an exhaustive list of these transformations and a representative list of enhancements 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 a rule's match fields); modifications to the rule's action list (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).

Controlling enhancements with SDN-Flags: Next, we discuss two SDN-Flags that Apps can use to control transformations that violate correctness or efficiency. Specifically, the transformations in discussed in § 2:

Input-Output dependence {IO}: This SDN-Flag specifies that the App's inputs are a function of the rules installed in the network (the Apps' output). This SDN-Flag allows the controller to ensure correctness of the Apps by circumventing

```
public interface Extensions {
  public ArrayList<Transactions> process_transaction
  (ArrayList<Transactions> );
  public void init();
  public void configure (HashTable<String,String>);
}
```

Figure 4: *Interface for enhancements.*

enhancements whose transformations lead to coarser granularity rules.

Push-Flag {PF}: This SDN-Flag allows the controller to directly perform the App's proposed changes into the network while simultaneously applying the enhancements to these actions. When the enhancement returns the optimized (transformed) rules, the controller replaces the App rules with the optimized version.

4.3 Mozart

In Figure 3, we present Mozart a redesign of the modern controller architecture that applies the compiler-optimization philosophy to SDN enhancements. Mozart exposes a novel interface to the Apps which enables Apps to bundle SDN commands into *transactional policies* (§ 4.1) and annotate the transactions with SDN-Flags (§ 4.2). The controller includes an orchestrator, similar to a compiler toolchain, that orchestrates enhancements, applies them to Apps, and ensures that SDN-Flags are respected. In Mozart, enhancements are integrated into the controller as isolated modules within the orchestrator: communication is through function calls or RPCs.

Interfaces: Mozart defines well-specified interfaces for Apps to interact with the controller and for smoothly integrating the enhancements into the Orchestrator.

The App interface, Figure 2, includes the API call that Mozart exposes to all Apps: `Apply()`. Using `Apply()`, an App can specify a `Transaction`, bundle of SDN messages, to apply to the network rather than individual messages. Furthermore, Apps may annotate these transactions with SDN-Flags— either one SDN-Flag for the entire transaction or a separate SDN-Flag for each message in the transaction.

The enhancement-interface, Figure 4, enables the orchestrator to manage enhancements and promotes interoperability between enhancements. To this end, the interface specifies a set of functions that each enhancement must implement. When the orchestrator initializes a new enhancement, due to a new DAG or modifications to an existing DAG, it calls the enhancement's init() function. As network operators modify the controller's configurations and alter an enhancement, the orchestrator calls `configure()` to reconfigure the enhancement. When an App calls `Apply()`, the orchestrator accepts the transaction and passes it through the set of enhancements listed in the DAG: the `process_transaction()` is called for each enhancement– the output of one

`process_transaction()` is used as input for the next `process_transaction()`.

Orchestrator: Runs within the controller and accepts an operator defined configuration: A linear DAG of enhancements to apply to a specific App. The Orchestrator accepts a transaction from an App, through the `Apply()` API call, determines the DAG for the App, and propages the transaction through the enhancements in the DAG. The output of the final enhancement (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 pre-specified SDN-Flags.

At a high level, the Checker verifies that no violating transformations as defined by the SDN-Flags specified through `Apply()` 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 they are applied, the Checker reverts the transaction back to its original state. When, the {PF} SDN-Flag is specified, the Orchestrator monitors the chain of enhancements and if the set of enhancement takes longer than 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 enhancements are done.

Using Mozart: In Mozart, the network operator specifies a linear DAG of enhancements to be applied to each App—the orchestrator uses this DAG to determine orchestration. The operator also specifies a list of enhancements that can not be avoided, e.g., *a security enhancement should have priority over SDN-Flags specified by any App.* The developer, writes Apps to leverage Mozart's interface and employs SDN-Flags when necessary.

5 PROTOTYPE AND EVALUATION

We developed a prototype implementation of Mozart including our interfaces and the orchestrator. Our prototype is built atop the Floodlight controller in 1326 Lines of Code (LoC). We changed Hedera to use SDN-Flags: {IO} for OpenFlow rules to edge switches and {PF} for all rules. We changed the TCAMOptimizer and the ConflictResolver to provide the functionality discussed in § 4.3.

Preliminary Results. We observe that applying Mozart improves bandwidth to within 98% of optimal: thus, improving Hedera's performance. Recall, the goal of the App is to maximize network bandwidth utilization while that of the TCAMOptimizer is to minimize TCAM usage. Although Mozart drastically improves Hedera's performance, we observe that Mozart reduces the efficiency of the TCAMOptimizer – the TCAMOptimizer only achieves a TCAM savings of 18.2% (instead of 57%). This trade-off between the effectiveness of the TCAMOptimization and the App's performance occurs because Mozart improves performance by limiting coalescing on OpenFlow rules. As part of future work, we will explore methods for systematically exploring these trade-offs.

6 RELATED WORKS

The most closely related works on SDN composition [5, 8, 18, 20] focuses on providing enhancements that promote principled composition of Apps with different objectives [5, 18, 20] or Apps running on different controllers [8]. Our work represents a fundamental departure from existing work in the composition space, rather than focusing on the Apps, we concentrate on the enhancements. Thus allowing us to introduce a similar level of rigor and understanding to enhancement-composition as we currently have for App-composition.

Orthogonal approaches at employing compiler techniques to SDN focus on enabling controllers to compile policies down to different south-bound APIs [14]; to effectively support network updates [29]; to compile monitoring specific functionality [21]; or, to efficiently compile high-level policies into rules [19]. These approaches are orthogonal, in that, they do not explicitly tackle enhancements or explore parallels between SDN and compiler optimizations.

Our abstractions represent a natural extension of Operating System hints [7, 11] to SDN's Network Operating System (controller).

7 CONCLUSION

This paper sheds light on the interactions between enhancements and Apps and, in doing so, highlights several troubling implications. Motivated by these implications, we argue for the design of a more powerful interface between the Apps, the SDN controller, and the enhancements– this interface allows for a systematic and principled inclusion of enhancements into the SDN ecosystem. Our design and prototype implementation of Mozart is the first step towards a holistic controller architecture capable of supporting enhancements in a manner that does not compromise the simplicity promised by SDNs (or the performance, and efficiency of the Apps). We believe this idea of a holistic controller architecture capable of integrating and composing enhancements presents a rich field of future research and grow in importance as SDN deployments continue to grow.

8 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 grant CNS-1409426.

REFERENCES

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

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[3] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In *CoNext*, 2014.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[5] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.

[6] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying Software-Defined Network Optimization Using SOL. In *NSDI*, 2016.

[7] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional Networking: Opportunistic Exploitation of Mobile Network Diversity. In *MobiCom*, 2010.

[8] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A Compositional Hypervisor for Software-defined Networks. In *NSDI*, 2015.

[9] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient Traffic Splitting on Commodity Switches. In *CoNEXT*, 2015.

[10] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT*, 2013.

[11] T. Karagiannis, R. Mortier, and A. Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *SIGCOMM*, 2008.

[12] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

[13] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.

[14] H. Li, C. Hu, P. Zhang, and L. Xie. Poster: Modular SDN Compiler Design with Intermediate Representation. In *SIGCOMM*, 2016.

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

[16] S. Luo, H. Yu, and L. Li. Practical Flow Table Aggregation in SDN. *Comput. Netw.*, 92(P1):72–88, Dec. 2015.

[17] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. T. Vechev. SDNRacer: detecting concurrency violations in software-defined networks. In *SOSR*, 2015.

[18] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner. Corybantic: Towards the Modular Composition of SDN Control Programs. In *HotNets*, 2013.

[19] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL*, 2012.

[20] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-defined Networks. In *NSDI*, 2013.

[21] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker. Compiling Path Queries. In *NSDI*, 2016.

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

[23] Project Floodlight. http://www.projectfloodlight.org/.

[24] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN*, 2013.

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

[26] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. L. Pacheco, J. Moulierac, and G. Urvoy-Keller. Too Many SDN Rules? Compress Them with MINNIE. In *GLOBECOM*, 2015.

[27] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-state Management Service. In *SIGCOMM*, 2014.

[28] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[29] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu. Compiling Minimum Incremental Update for Modular SDN Languages. In *HotSDN*, 2014.

[30] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *SIGCOMM*, 2010.