Let me rephrase that: Transparent optimization in SDNs

Santhosh Prabhu* Mo Dong* Tong Meng P. Brighten Godfrey Matthew Caesar
University of Illinois, Urbana-Champaign
{prabhum2, modong2, tongm2, pbg, caesar}@illinois.edu

ABSTRACT

Enterprise networks today have highly diverse correctness requirements and relatively common performance objectives. As a result, preferred abstractions for enterprise networks are those which allow matching correctness specification, while transparently managing performance. Existing SDN network management architectures, however, bundle correctness and performance as a single abstraction. We argue that this creates an SDN ecosystem that is unnecessarily hard to build, maintain and evolve. We advocate a separation of the diverse correctness abstractions from generic performance optimization, to enable easier evolution of SDN controllers and platforms. We propose Oreo, a first step towards a common and relatively transparent performance optimization layer for SDN. Oreo performs the optimization by first building a model that describes every flow in the network, and then performing network-wide, multi-objective optimization based on this model without disrupting higher level correctness.

CCS Concepts

•**Networks** → **Network architectures;**

Keywords

Software-Defined Networking, Optimization

1. INTRODUCTION

Simplification of network management is not an easy goal to achieve, especially with diverse requirements that administrators have of their networks. Typically, these requirements fall in two categories: *correctness* and *performance*. Correctness (more accurately functional correctness) defines reachability, access control and chaining between different types of endpoints and services. Performance, used in a broader sense here, includes optimization, resilience and monitoring.

Though correctness may appear to simply boil down to "what can talk to what through what", the higher level policy abstractions that network administrators require to define correctness are extremely diverse. The definition of correctness, and the abstraction used to state it depends on many factors, such as the nature of the enterprise, traffic the network carries, applications that use the network, various compliance requirements, whether the network is a private data center or a campus or a WAN, etc. Furthermore, the active SDN research community is now experimenting with novel abstractions. Performance on the other hand is better understood, and there exist well-defined best practices to meet the typical performance goals that apply to many enterprises. In fact, nearly all enterprises outside the advanced technology sphere rely on a smaller set of generic off-the-shelf features for performance (shortest path routing, ECMP, and MPLS-TE being examples).

We believe that SDN can and should be an enabler for creating diverse higher level correctness abstractions that match the requirements of enterprise networks. In fact, there do exist OpenFlow-based SDN platforms that define a variety of specification abstractions. However, they all bundle correctness and performance together as a single problem, and define a common abstraction to manage both. For example, FatTire [15] provides an abstraction that combines a regex-based path specification with a fault tolerance requirement. SIMPLE [14] combines service chaining with switch memory optimization. The SOL [7] API aims at simultaneously managing both correctness and performance.

This monolithic architecture has some major drawbacks. Given the sheer diversity of correctness goals, having unified platforms for correctness and performance will require an ecosystem of monolithic platforms, each managing both correctness and performance. Such an ecosystem would take a formidable amount of effort to create – in addition to the diversity of correctness goals, network optimization is a non-trivial challenge that in itself requires significant effort to tackle. Moreover, evolving and maintaining control platforms over time is more difficult if they are monolithic. For enterprise users, combining the artifacts of correctness and performance significantly hinders agility, since adoption of new techniques may require a complete overhaul of existing network management mechanisms.

To overcome the limitations of such coupling, in this paper, we argue that correctness and performance should be treated as modules that can evolve separately. We envision a rich set of correctness abstraction platforms that focus on policy constructs attuned to enterprise needs, each producing a **correct** data plane in OpenFlow (or a future standardized format). A separate performance module then takes this

*Author order determined by a coin toss

SOSR '17, April 03-04, 2017, Santa Clara, CA, USA
 ISBN 978-1-4503-4947-5/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050220.3050226

correct data plane and generates an optimized data plane. Moreover, leveraging the relatively generic and well understood performance needs across many network scenarios, a commonly-applicable performance module can be built requiring minimal configuration.

In this paper, we present Oreo as first step towards a common performance optimization platform to enable decoupled correctness and performance management. The concept of Oreo is similar to program optimization features provided by compilers. Just as optimizations like loop unrolling and code inlining can be done without the knowledge of the programmer, in SDNs, path shortening, rule compression, etc. require no operator interaction. And just as compilers can perform these optimizations on programs of every conceivable nature, the SDN optimization layer can be largely agnostic to higher level correctness constructs.

To that end, Oreo takes a *correct* data plane computed by any upper layer as input and then constructs a network-wide data plane *flow model* describing the forwarding behavior of the entire network. This flow model also gives Oreo the capability to perform network-wide multi-objective performance optimization unlike many single-device optimization systems previously proposed [4, 12]. We present a proof of concept of our system which focuses on transparent network-wide static optimization with multiple objectives – minimizing number of forwarding rules and average path length. We conclude with a future research roadmap for a dedicated performance enhancement layer.

2. A CASE FOR DECOUPLING CORRECTNESS AND PERFORMANCE

As discussed in §1, correctness in enterprise networks constitutes a diversified and complex space. In modern software-defined data centers, administrators set the security policy of application-based end-point groups in the face of constant change such as VM migration and dynamic application provisioning. In more organic networks like campus networks, correctness requirements do not have broad standardization. A financial institution might need to isolate PCI-compliant payment processing networks from operational networks; a retail store might need to segment customer Internet access and the credit card payment network. Many organizations from universities to manufacturing plants have DMZs that protect secure sites and at the same time bridge outside control commands. In traditional network settings, the network architect's biggest challenge is to accurately translate the higher level correctness intent (such as all the examples above) to lower level implementation involving numerous technologies such as physical attachment points, VLAN assignment, IP subnetting, VRFs, route filtering and black-holing, firewall configurations, middlebox placement, etc. This process is challenging and involves hiring armies of network operators.

On the other hand performance configuration is relatively more generic. Certain particular industries require special advanced technology (like ISPs doing advanced traffic engineering or algorithmic trading firms optimizing microseconds of latency), but most enterprises are satisfied with over-provisioning or using common features like shortest path routing, use of multiple equal-cost paths (via ECMP at layer 3 or port-channels at layer 2), and occasionally certain MPLS features (fast reroute).

SDN presents an opportunity to create platforms that network operators can use to define diverse high level correctness requirements without worrying about that translation process. This is exemplified by the recent commercial successes of network virtualization products like Cisco ACI[3] and VMware NSX[17], which are designed to provide convenient abstractions for administrators to manage policies in data centers. This also shows it is common for enterprise networks to focus on managing correctness, with much less fine-grained management of performance.

In the SDN research community, the SDN management platforms that do offer higher level correctness abstractions have invariably coupled them with performance management. For example, FatTire [15] provides an abstraction that combines a regex-based path specification with a fault tolerance requirement. Merlin [16] combines service chaining with switch memory optimization. These may be difficult for network operators to use, since they must jointly specify both correctness and performance goals. SOL [7] provides a library API that controller authors can use to implement a variety of correctness abstractions and performance optimizations. This may accelerate the process of implementing controllers since it effectively allows some optimizations to be shared across different controllers. But most fundamentally, all of the above controllers are *built as a monolithic unit trying to accomplish both performance and correctness.* For that approach to be successful, the SDN research community either has to (1) build a large collection of SDN platforms, with support for various combinations of correctness and performance abstractions, or (2) design a universally acceptable abstraction for all enterprises. Given the diversity of requirements, both these approaches are fundamentally hard. Even with libraries like SOL available for platform creators to implement their correctness abstractions, the challenges involved in creating, maintaining and evolving a multitude of optimization platforms makes the first approach onerous. The second approach is similarly hard, because a single abstraction will have difficulty matching the multitude of correctness specifications that exist in the enterprise space.

We believe the right solution to the problem lies in decoupling correctness from performance. The correctness layer can focus on allowing the network manager to specify high level intent; performance can be optimized in a lower layer, as transparently as possible and requiring only simple configuration from network managers, as in traditional networks. A common communication format (like OpenFlow) can be used for easy interaction between various correctness platforms and the single performance management layer.

This layered architecture has several benefits. Most importantly, it will lead to a **more flexible SDN ecosystem and accelerate innovation**. A diversity of different correctness-oriented controllers can cater to diversified requirements as described earlier. The performance layer can then incorporate optimizations which may be basic or may be advanced

– optimizations of flow routing, resilience, table size, etc. – but are generally relatively generic, so they can benefit many higher-layer controllers. (Indeed, truly transparent optimizations can even benefit controllers that have already been developed!) In addition, we believe there are significant **network management benefits:** a single performance layer could support multiple controllers in a single deployment, simplifying operations; separating correctness and control with an open API between them provides a powerful monitoring point to improve visibility and isolate problems to one layer or the other; and separate software could align better with separate teams in some enterprises (e.g. security team managing the correctness controller, network team managing the performance controller). Separating the two controllers also allows enterprises to modify/upgrade one independent of the other, promoting greater agility within the enterprise. Note that these advantages do not apply to monolithic platforms, including those built using SOL.

To push towards this architecture, we focus on the common performance layer, called Oreo. Oreo takes a correct data plane computed by any upper layer as input and then constructs a network-wide data plane flow model describing the forwarding behavior of the entire network. This flow model is the foundation for Oreo's network-wide performance enhancement. First, from this model, Oreo can automatically infer the constraints for optimization – the end-to-end reachability/isolation of packets and service chaining. This is the key to keeping Oreo transparent. Second, this flow model gives Oreo the capability to perform network-wide multi-objective performance optimization that differs from previous single-device optimization systems (e.g. single-device rule compression [4, 12]). The scope of Oreo is to enhance the following aspects of data plane: (a) Static optimization, e.g. network-wide rule compression, forwarding path length reduction, and dropping blacklisted traffic early; (b) Resilience, e.g. single-link failure resilience with automatic backup rule computation; (c) Dynamic traffic optimization, e.g. traffic engineering using active traffic measurement and Oreo's flow model.

3. SYSTEM ARCHITECTURE

Fig. 1 illustrates Oreo's proposed architecture. Oreo expects a layer above it to specify a *correct* data plane, and builds a network-wide flow model describing end-to-end paths along which traffic is forwarded. It then uses a multi-objective optimization mechanism to determine the best paths and finally pushes data plane rules into the switches. We now describe each component of Oreo in greater detail.

Network Modeler directly receives OpenFlow data plane rules and network topology output from the SDN controllers to create and update a network-wide flow model. Essentially, the network model is a correctness declaration from the upper layer and defines the paths and final fates of any packet injected at any interface in the network. This model is defined based on *equivalence classes*, which are collections of packets whose behavior is exactly identical throughout the network. A combination of an ingress device interface and an EC defines a forwarding path and that is the atomic unit

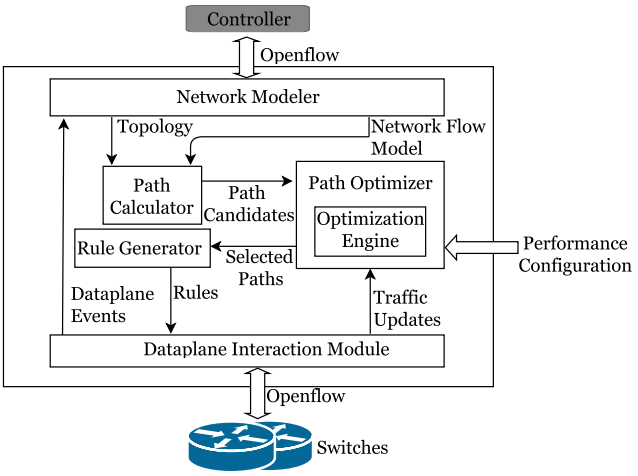


Figure 1: Oreo Architecture

for Oreo to perform optimization. The capability of building this network model from OpenFlow data plane rules makes Oreo transparent to the correctness layer.

Path Calculator's task is to compute candidate paths that are going to act as feasible values of decision variables during optimization, for all (EC, ingress interface) pairs. Based on the network model, Path Calculator automatically figures out the constraint on the alternative paths: packets forwarded on them should have the same end fate as on the original paths defined by the correctness layer. Service chaining is also maintained by treating middleboxes as both path termination and initiation points, meaning that traffic is assumed to leave the network and re-enter. The choice of algorithm for calculating the candidate paths depends on the type of performance optimization involved: stretch reduction naturally favors shorter paths and fault tolerance on the other hand favors disjointedness of paths. We expect that good heuristics can help in having a good combination of paths being selected as candidates. For this proof of concept, we use Yen's algorithm for k-shortest paths [18].

Path Optimizer module performs network-wide, multi-objective performance optimization by using the network flow model and picking the best alternative paths from among those provided by Path Calculator. We use several examples from different categories to illustrate the scope of Oreo's performance optimization.

First, static optimization focuses on reducing forwarding paths and network resources like switch memory. We discuss two examples here. The first example is Network-wide rule compression. The scope of Oreo's switch rule compression is much wider than existing rule compression platforms. Thanks to the network-wide flow model, Oreo understands how any traffic flows through the network and therefore understands what rules can *actually* be hit by traffic. So, post-optimization, only rules which can be hit will be pushed into the dataplane. In addition, Oreo can also change end-to-end EC paths, such that more paths share same rules, to save switch memory even further. The second example is path optimization. Naturally, Oreo's understanding of network-wide flows allows it to choose shorter paths with equivalent behavior. An interesting special case for path optimization

is that Oreo can drop unwanted traffic as early as possible to reduce wasted traffic in the network.

Second, resilience proactively and reactively addresses failures. Oreo can proactively compute backup rules and pre-install them to OpenFlow group tables. Oreo can also react dynamically to link failures and install alternative rules (pre-computed or computed reactively) to maintain the original behavior of the flows.

Third, dynamic traffic optimization combines the knowledge of the flow model and active traffic measurement to alleviate hotspots in the network when possible. While the traffic engineering optimization can be using new or existing mechanisms, we believe that leveraging the network wide model for TE can pay rich dividends particularly in less structured topologies (outside the datacenter).

Due to the multi-objective nature of the optimization problem, Oreo would need to ultimately expose simple knobs similar to today's compiler optimization options to end users to express weight on different optimization goals. As a proof of concept for the network-wide multi-objective performance optimization, in this paper, we only focus on formulation and evaluation of the two static optimization goals listed above.

Rule Generator takes the EC paths output by Path Optimizer and translates these paths into actual rules to be inserted into the data plane. This process is needed because during the optimization process, we may not be able to always compute the actual rules, but rather use heuristics, like in [7], to represent different optimization objectives. Ideally, the performance optimization step should combine rule computation and path selection into a single optimization problem. Prior work on optimized rule placement can be leveraged [8] in computing the optimal set of rules that accurately enforce the computed state. The rule computation will work on a similar flow table layout used in commercial forwarding devices, where each table is responsible for a particular function, including VLANs, L2 forwarding, L3 unicast, L3 multicast etc.

Data Plane Interaction Module installs the computed rules into the switches and is also responsible for monitoring network events and traffic when needed.

4. PROOF OF CONCEPT

To evaluate the feasibility and usefulness of Oreo, we create a proof-of-concept path optimizer module that focuses on two static optimization goals: minimizing path length of packet forwarding and minimizing the switch memory usage. We use network model generated using the VeriFlow dataplane verifier [11] and path candidates picked using a k-shortest path algorithm as input for this Path Optimizer module. We chose Integer Linear Programming as our optimization technique, primarily due to its proven success in network dataplane optimization [7, 14, 16]. Next, we describe the formulation of the ILP.

We represent the set of all the network devices as:

$$\{D_i \mid i = 1, 2, \cdots, M\}.$$

Device D_i is assumed to contain N_i flow tables. For ease of exposition, we uniformly number all the $N = \sum N_i$ forwarding tables by defining the set of all flow tables as:

$$\{TB_j \mid j = 1, 2, \cdots, N\}.$$

The capacity of TB_j, in number of rules, is c_j. Then, the set of equivalence classes to be optimized is denoted by:

$$\{EC_k \mid k = 1, 2, \cdots, K\}.$$

Each EC_k contains R_k ingress-egress pairs, which determine a set of flows through the network. Assume that for the ingress-egress pair r in EC_k, there exist S_r^k paths available by the Path Selector. We name these paths P_{ri}^k $(0 \leq i < S_r^k)$, where P_{r0}^k always denotes the path defined for EC_k by the controller. The length of path P_{rs}^k is denoted by h_{rs}^k. We also define, for each path P_{rs}^k, a set of *binary forwarding flags*, $x_{rs}^k(i,j)$, denoting the existence of packet forwarding from TB_i to TB_j when traffic flows along P_{rs}^k.

To solve the optimization problem, the Path Enhancer module needs to assign values to decision variables v_{rs}^k, indicating whether P_{rs}^k is selected or not. The objective function to be maximized is computed as the sum of the reduction in path length and the amount of switch memory saved. The terms are weighted by coefficients a and b, respectively. The reduction in path length is computed as the difference between the original path that is defined by the controller and the new path that is chosen, summed over all equivalence classes. Similarly, switch memory reduction at each table is calculated as the reduction in the number of ECs traversing it. This is a heuristic that has been shown to work well [7], but we do not rule out choosing other heuristics if needed. Naturally, changing the coefficients a and b causes the computed data plane to vary. A higher a favors selection of shorter paths, whereas a higher b causes paths to be shared by multiple ingress-egress pairs.

Mathematically, the objective function is as follows:

Maximize:

$$a \cdot \sum_{k=1}^{K} \sum_{r=1}^{R_k} \sum_{s=1}^{S_r^k} v_{rs}^k \left(h_{r0}^k - h_{rs}^k\right) + b \cdot \sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[sgn\left(\sum_{r=1}^{R_k} x_{r0}^k(i,j) \right) - z_{ij}^k \right]$$

The following constraints need to be satisfied:

- Forwarding table TB_i has at most c_i forwarding rules.
- Each ingress/egress device pair uses only one path.
- Each EC_k either doesn't reach forwarding table TB_i, or matches exactly one rule (*i.e.*, a unique next hop) in TB_i.

These constraints can be written precisely as:

$$\sum_{k=1}^{K} \sum_{j=1}^{N} z_{ij}^k \leq c_i, \quad i = 1, \cdots, N$$

$$\sum_{s=0}^{S_r^k} v_{rs}^k = 1, \qquad k = 1, \cdots, K \quad r = 1, \cdots, R_k$$

$$\sum_{j=1}^{N} z_{ij}^k \leq 1, \qquad k = 1, \cdots, K \quad i = 1, \cdots, N$$

$$v_{rs}^k \in \{0, 1\}, \qquad k = 1, \cdots, K \quad r = 1, \cdots, R_k$$

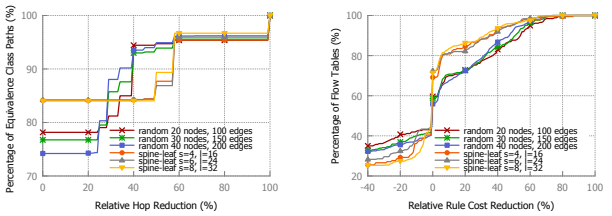
$$s = 0, 1, \cdots, S_r^k$$

$$\frac{1}{R_k} \cdot \left[\sum_{r=1}^{R_k} \sum_{s=0}^{S_r^k} v_{rs}^k \cdot x_{rs}^k(i,j) \right] \leq z_{ij}^k.$$

$$i, j = 1, \cdots, N \quad k = 1, \cdots, K$$

5. EVALUATION

We implemented a proof-of-concept Path Optimizer module using Gurobi [6] to minimize switch memory cost and path stretch. In our experiments, we evaluate the path optimizer's feasibility in terms of optimization performance and result quality. We also examine the effect of the multi-objective nature of this optimization. For different synthesized topologies, we assume each device hosts two IP subnets and has the following flow table pipeline: each interface has a per-interface ACL table that drops traffic; and traffic that is not dropped by the ACL table goes to a central routing table that decides the next hop. To generate rules on each table, we first compute shortest path routing rules with 30% of the links masked out. This results in non-shortest path forwarding in the network. For each interface's ACL table, we pick a random number (uniformly distributed between 0 to 5% of number of prefixes) of ACLs, with each ACL dropping an random IP prefix in the network. With the synthesized data plane, we use Veriflow [11] to generate the network flow model. The path calculator was configured to determine 5 shortest paths for each ingress-egress pair as the candidate set.



(a) Relative Hop Reduction (b) Relative Rule Cost Reduction

Figure 2: Static Optimization Performance from Scratch

One-shot Optimization Performance: We conduct optimization from scratch for topologies of different structure and size, with $a = 0.5$ and $b = 0.5$. For each device, we compute the percentage of reduction in rule cost after optimization. For each equivalence class at each ingress interface, we compute the percentage reduction in hop count after optimization. Fig. 2 plots the CDF of relative reduction in the rule costs and hop counts (We only show the rule-cost reduction for flow tables that originally had at least one rule). 15% to 25% of the flows see a reduction in path length, and 60%-70% of the flow tables have smaller memory utilization, demonstrating that the formulation and optimization is promising and effective in this data set.

Fig. 3 compares the time taken for optimization in different topologies. Naturally, the optimization is slower in larger topologies with more ECs. For the largest topology, 8 spines and 32 leaves with 989 ECs and 672 flow tables, the computation time is $19s$. We believe this preliminary benchmark on the synthesized network suggests that Oreo is feasible for optimizing larger networks too. Moreover, in practice, the network-wide optimization from scratch can be allowed more time as compared to smaller incremental updates, which we evaluate next.

Low Latency Incremental Update: Transparent SDN solutions must respond to network updates in real-time. In the case of Oreo, latency may be incurred in one of two stages: computing an updated network model and optimizing the updated model. It is known that computing a network model can be achieved in real time [10, 11], so we focus our evaluation on the optimization time. We expect Oreo to be naturally fast for small updates, since it optimizes one EC at a time, and most single OpenFlow updates will cause only a small number of ECs to change their behavior. To verify this idea, we ran the following experiment. After computing the optimized flow paths for all equivalence classes, we redefine ingress-egress behavior for one equivalence class, and measure the time taken to recompute the optimized paths for those packets. Figure 4 shows the CDF for the time taken across all ECs. All updates are computed in less than 40 milliseconds, with the worst median update time smaller than 17 milliseconds. That illustrates Oreo's capability to support low-latency updates. If necessary, latency of Oreo can be further improved by first installing a suboptimal configuration (possibly the one computed by the controller), and subsequently replacing it with an optimized version.

Multi-Objective Tradeoff: To evaluate the multi-objective optimization trade-off described in § 4, we assign different values to a and b in the formulation. Fig. 5 shows the trade-off between the two objectives by tuning the path stretch weight a ($b = 1 - a$) on a random graph topology with 30 nodes and 150 links. As a increases, Oreo favors shorter paths with less hops, at the cost of generating more forwarding rules. To the extreme case of $a = 0.1$, i.e., heavily optimizing switch memory costs, we observe that many forwarding paths that are 2 and 4 hops longer than before.

6. DISCUSSION

Managing Performance: By separating performance from correctness, Oreo frees the upper layers from the burden of performance management. So, tuning of network performance shall be done by the network operators interacting directly with Oreo. While a complete design of such a configuration mechanism is to be addressed in future, we observe that it should allow most administrators to simply choose from a set of well defined configuration options, or, in some more advanced cases, specify QoS classes for Traffic Engineering. Even in cases where operators perform finer-grained performance management, a separate management mechanism will facilitate easier administration, due to the separation of concerns for Openflow-based correctness management abstraction.

The Openflow Abstraction: Oreo's choice of Openflow as the northbound abstraction may appear counterintuitive, especially since Oreo only attempts to preserve reachability characteristics. The "One-big-switch" abstraction would appear a more natural fit. We choose Openflow purely because it is more universally accepted, and allows Oreo to be used even with existing controllers. In future, when a standard-

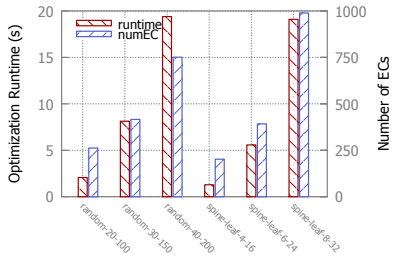


Figure 3: ***Time taken for Optimization from Scratch***

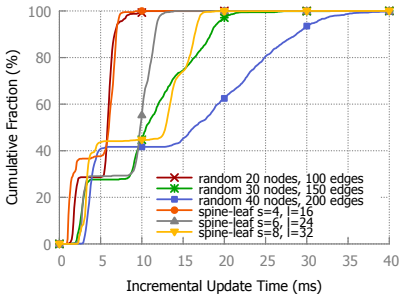


Figure 4: ***Time taken to update one Equivalence Class***

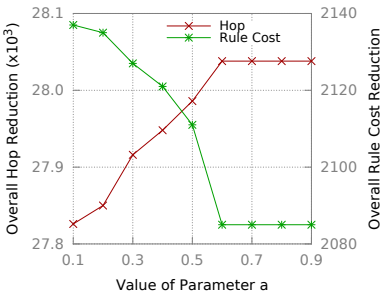


Figure 5: ***Effect of Different Optimization Objective Weights***

ized reachability abstraction is defined, Oreo could be used with minimal changes to work with the new abstraction.

It may be noted that even though Oreo is designed to work with Openflow, it doesn't preserve full Openflow semantics, one example being dataplane counters. We believe this is justified, since counters are typically not relevant to a correctness specification, which Oreo tries to preserve. Nevertheless, it is possible to emulate dataplane counters in Oreo, by using *alias counters* in the optimized dataplane.

Being completely transparent to higher and lower layers, Oreo can be used in any standard OpenFlow network. Oreo can also be used in conjunction with other transparent OpenFlow enhancement tools like [9]. However, for the combination to be productive, some intelligent architecting may be warranted. For example, switch memory management mechanisms should be placed closer to the dataplane than Oreo, so that they can meaningfully provide their function.

7. RELATED WORK

Fabric[2] is an alternative SDN architecture that is also aimed at spearation of controller concerns. Fabric advocates a *horizontal* separation between the edge (such as hypervisors in a datacenter, responsible for correctness) and the core (providing connectivity with performance). In contrast, Oreo relies on a *vertical* separation, and looks at both correctness and performance from a network-wide perspective, without any distinction between the edge and the core. We believe that Oreo's approach to layering may fit better with enterprise networks where correctness and performance are not cleanly partitioned into the edge and the core.

Comparison of existing declarative OpenFlow programming frameworks, such as Merlin [16], FatTire [15] and SIMPLE[14], is already discussed in §1 and §2. At first glance the architecture of an Oreo-based SDN platform may look strikingly similar to one where a compiler like NetKAT[1] is used to translate higher level specifications into dataplane state. One example of such a layered architecture was proposed for compiling path queries into dataplane rules[13]. Oreo is different from such architectures in that it performs optimizations with a much greater scope, since only the end-to-end reachability characterstics need to be preserved. Moreover, Oreo is more than a simple translator — it actively monitors the dataplane and modifies it in response to performance-related events.

Performing single-device optimizations on data plane configurations has been quite well studied, mostly in relation to compressing firewall rules or routing tables to conserve switch memory. Work in this space includes Firewall Compressor [12], Diplomat [4] and Optimal Routing Table Constructor (ORTC) [5]. These tools optimize only switch memory, and only locally at a device, lacking a network-wide view. Thanks to such a view, Oreo can perform a more diverse set of optimizations, and even in terms of switch memory, achieve better compression, by eliminating redundancy across the network rather than at a single device alone.

CacheFlow[9] is a rule caching mechanism that allows multiple switches to behave as a single switch with infinite memory. Oreo and CacheFlow are somewhat orthogonal in that CacheFlow does not attempt to change the data plane, whereas Oreo recomputes a new data plane with equivalent semantics. They can be used simultaneously, or caching functionality could be folded into Oreo.

8. CONCLUSION

In this paper, we argued why a clear separation of correctness and performance goals can allow controllers to focus on correctness and abstract representation, while providing more universal optimization capabilities. Specifically, Oreo is a generic and transparent layer which optimizes network performance while preserving data plane equivalence.

Future work should build Oreo into a fully-featured dataplane optimization platform, capable of both static and dynamic optimization, in terms of both performance and robustness. We expect to leverage the rich collection of existing work to tackle various challenges, including path selection, rule placement etc., but joining these into a multi-objective optimization will require new solutions. A longer-term expansion of our architecture could continue the analogy to compilers: where a compiler like LLVM uses an intermediate representation and can produce machine code for many target platforms, Oreo could treat OpenFlow as an intermediate representation and produce data plane instructions for non-OpenFlow switch hardware (e.g. a collection of MPLS tunnels with appropriate ingress/egress processing). More important than any one optimization, we believe our improved modularity of the controller ecosystem can pay substantial dividends to the SDN community.

This work was supported by NSF CNS Award #1513906 and by the Maryland Procurement Office under Contract No. H98230-14-C-0141.

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1):113–126, Jan. 2014.

[2] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 85–90, New York, NY, USA, 2012. ACM.

[3] Cisco Systems Inc. Cisco Application Centric Infrastructure Zero-Touch Fabric, 2013.

[4] J. Daly, A. X. Liu, and E. Torng. A difference resolution approach to compressing access control lists. *IEEE/ACM Transactions on Networking*, 24(1):610–623, Feb 2016.

[5] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing Optimal IP Routing Tables. In *In Proc. IEEE INFOCOM*, pages 88–97, 1999.

[6] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual, 2015.

[7] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying Software-Defined Network Optimization Using SOL. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 223–237, Santa Clara, CA, Mar. 2016. USENIX Association.

[8] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[9] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, New York, NY, USA, 2014. ACM.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, 2013. USENIX.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 15–28, Berkeley, CA, USA, 2013. USENIX Association.

[12] A. X. Liu, E. Torng, and C. R. Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, April 2008.

[13] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, Santa Clara, CA, 2016. USENIX Association.

[14] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[15] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[16] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster. Managing the network with merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 24:1–24:7, New York, NY, USA, 2013. ACM.

[17] VMWare Inc. The VMware NSX Network Virtualization Platform, 2013.

[18] J. Y. Yen. An Algorithm for Finding Shortest Routes from All Source Nodes to a Given Destination in General Networks. *Quart. Applied Math*, 27:526–530, 1970.