Intent-Driven Composition of Resource-Management SDN Applications

Victor Heorhiadi
University of North Carolina
Chapel Hill, NC, USA
victor@cs.unc.edu

Sanjay Chandrasekaran
Carnegie Mellon University
Pittsburgh, PA, USA
sanjayc@andrew.cmu.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Vyas Sekar
Carnegie Mellon University
Pittsburgh, PA, USA
vsekar@andrew.cmu.edu

ABSTRACT

As software-defined networking deployments mature, operators need to manage and compose multiple resource-management applications, such as traffic engineering and service chaining. Today such applications' resource management algorithms run separately and composition approaches are output-driven, e.g., running each application on a statically provisioned slice of the network and then combining the flow rules output for each slice. Such approaches result in inefficient resource utilization and unfairness. Instead, we argue for intent-driven composition, where a unified resource optimization formulation is composed from applications' high-level intents and the solution to this problem formulation is realized in the network. We design Chopin[1], an intent-driven framework for composing SDN resource-management applications. Chopin's design addresses key robustness challenges with regard to efficiency and fairness that arise in realizing such an intent-driven approach. We have integrated Chopin with the ONOS controller and show that it substantially improves efficiency and fairness over existing composition approaches.

CCS CONCEPTS

• **Networks → Network resources allocation**; **Network management**;

ACM Reference Format:
Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. 2018. Intent-Driven Composition of Resource-Management SDN Applications . In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3281411.3281431

[1] Allusion to Frédéric Chopin, a 19th century classical composer

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281431

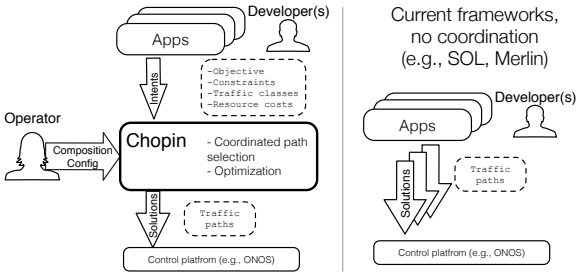

Figure 1: Comparison of Chopin framework vs. existing application deployment processes.

1 INTRODUCTION

As software-defined networking (SDN) transitions from infancy into the next stage, we see more complex deployments with multiple network management applications. Organizations have reported concurrently deploying multiple specialized applications on their networks [13, 46] and there are also visions of SDN "app stores" [34, 40, 41].

With the growing number and diversity of applications running in a network, there is an imminent need for systematic approaches for *composing* such application instances. While policy composition has been studied extensively and many solutions are available [22, 36, 38], composing resource-management applications remains a hard problem due to the wide range of applications and their diverse demands (e.g., load balancing, service chaining, power saving) [18, 29, 37]. Tensions among various types of resources (e.g., compute, bandwidth, latency) introduce additional complexity to SDN application composition, which has not been addressed as extensively as the policy composition problem.

Existing approaches for composing resource management requirements [4, 5, 22] take an *output-driven* approach, in which each application uses its respective standalone optimization algorithm to generate a candidate solution. An application's candidate solution might then be used to influence the resources made available to another application (e.g., as in ordered optimization) or be combined with outputs of other applications (e.g., statically slicing the network resources across applications). Some of these approaches even require applications to be rewritten to implement additional

composition logic (e.g., voting [4, 5]), but still produce suboptimal outcomes. As such, these output-driven approaches are far from optimal in terms of resource efficiency and/or fairness across applications.

In contrast to these existing approaches, we argue for an *intent-driven* composition approach as shown in Fig. 1. At a high-level, instead of executing multiple applications and deploying their output strategies directly to the network, intent-driven composition utilizes applications' high-level goals (or intents) to directly construct a unified optimization. An intent-driven approach has multiple benefits over output-driven composition (§2). First, it can produce a more resource-efficient solution by modeling the demands of all applications simultaneously, as opposed to executing applications separately. Second, it enables the operator to execute flexible and fair composition of applications, and systematically balance (possibly competing) objectives of multiple applications. Third, the applications do not need to be (re)designed to be composition-aware and need only to specify their own intents.

To this end, we develop Chopin, a framework that enables composition of resource-management SDN applications and addresses each of the challenges mentioned above. To provide expressive and general *intent specifications* for a broad spectrum of applications, Chopin builds on prior work called SOL [21]. SOL is conceptually similar to other related frameworks such as Merlin [44] and DEFO [16]. Essentially, these frameworks provide high-level APIs to express the requirements of various SDN apps—e.g., what kinds of packet processing need to be implemented and the latency/bandwidth requirements of different classes of traffic. For instance, using SOL we can specify that all HTTP traffic from prefix A to prefix B (i.e., a given class) needs paths of at most 5 hops (i.e., a performance requirement) and that this traffic must be processed by a firewall and an IDS in that order (i.e., packet processing or service chaining goals).

However, these frameworks, including SOL, are designed to capture the requirements of *only one* application at a time. As such, SOL's design, and the designs of these other related efforts, do not support composition, and naively extending them incurs the same pitfalls of the output-driven composition approaches discussed above.

Chopin uses the application intents (i.e., written using the SOL or a similar API) to *directly* construct a unified optimization, rather than per-application optimizations. This enables coordination among application demands without requiring the developers to create composition-aware applications, and produces a resource-efficient solution. Doing so also enables Chopin to support multiple candidate fairness metrics (e.g., [3, 15, 25]), providing flexibility in how the application objectives are balanced.

To produce an efficient solution at timescales that are sufficiently responsive to network and traffic changes, Chopin decomposes the solution generation into two steps: offline pre-processing that selects a subset of network paths over which to route application demand, and an online optimization that balances the traffic across these paths based on the current demand. While the use of offline-online decomposition for scalability is similar to prior efforts [16, 21, 28, 44], subtle issues arise from our need to consider a set of coexisting applications rather than a standalone application. For instance, if all applications greedily pick the same set of paths in the offline stage, then we could congest the network even though more optimal solutions are possible. Thus, we need the offline computation to be *coordinated* across the set of applications and tolerant to the variability in traffic demands across these applications. While this makes the offline computation more complex than the standalone case in SOL, we develop heuristics to improve its tractability.

We have implemented a Chopin prototype using Python and a Chopin service in Java for the ONOS controller (the source code is publicly available on Github [19]). The results indicate that Chopin achieves better optimality than naive composition approaches (e.g., static resource allocation) by as much as 40%. Chopin also outperforms composition based on voting mechanisms [5] in resource efficiency by a factor of 2. Our heuristics for path selection achieve an order of magnitude speedup while sacrificing $\approx 1\%$ optimality.

Contributions and Roadmap. To summarize, our contributions include the following:

- We empirically demonstrate the limitations of composing SDN resource-management applications using output-driven approaches (§2);
- We introduce intent-driven composition (§3) as an alternative, leveraging the applications' high-level goals available to intent-based frameworks;
- We achieve scalable composition with the help of traffic clustering and coordinated path selection (§4) and integrate with a popular SDN platform (§5).

We empirically evaluate our implementation in §6. We follow with discussion and limitations in §7, related work in §8, and conclude in §9.

2 BACKGROUND AND MOTIVATION

In this section, we highlight the limitations of existing output-driven composition approaches. At a high level, we refer to an approach as output-driven if the optimization algorithms for each application are executed separately and their outputs are either combined or used to influence inputs of other applications. Moreover, the optimization logic of the application is not known to the orchestration/composition framework. In effect, existing intent frameworks [16, 21, 44] would be forced to use output-driven composition to support multiple concurrent applications.

As a concrete example, consider Fig. 2. Suppose the network operator desires to install two different applications: one to balance the load that web traffic imposes on network links, and another to ensure that SSH traffic traverses a firewall and, subject to this constraint, travels minimal-latency paths. For clarity we examine only traffic traveling between nodes N_1 and N_5; it is easy to see that the optimal solution is for App_1 to use path N_1-N_2-N_4-N_5 and for App_2 to use path N_1-N_3-N_5. We use this example to discuss common output-driven approaches:

Static allocation [43] divides the resources, and each application is presented with a view of a topology based on those allocations. The allocations are computed proportionally to application priority (e.g., the amount of traffic that belongs to the application). For example, in Fig. 3, resources are divided proportionally by traffic volume, where App_1 perceives links to have $\frac{2}{3}$ of their physical capacity,

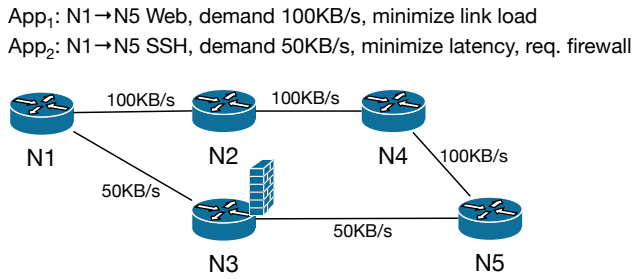

Figure 2: Composition scenario: two applications to be deployed and their optimizations to be applied to Web and SSH traffic (respectively).

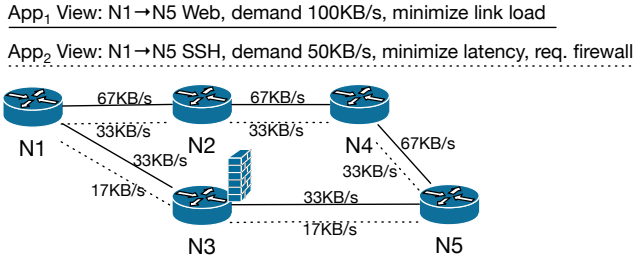

Figure 3: Static resource allocation provides an over-constrained view of links, resulting in a failure to enforce the firewall policy and sub-optimal latency for SSH traffic.

while App_2 perceives links with $\frac{1}{3}$ of their capacity. When the optimizations are executed, due to link constraints, $\frac{1}{3}$ of SSH traffic is forced to take the path N_1-N_2-N_4-N_5, which lacks a firewall and is longer than N_1-N_3-N_5 — resulting in a failed policy enforcement and suboptimal latency.

Ordered optimization [10] orders the applications and solves the optimizations in sequence. That is, after the first optimization is executed, the capacity of the network is adjusted by subtracting the resources it consumed and the next application's optimization can only use the residual capacities. In our example, after App_1 is run, due to link load-balancing, the residual capacity of the network is identical to that of App_2's view in Fig. 3: links with capacity 17KB/s and 33KB/s. This capacity is insufficient to correctly route the SSH traffic. While simply re-ordering the applications can alleviate this problem, for larger number of applications exploring all possible orderings to find the best solutions is impractical.

Voting schemes make improvements to the ordered optimization strategy. Examples include systems such as Corybantic [4] and Athens [5], which make applications *aware* of the other applications and allow them to vote on each others' resource-management proposals to negotiate a solution. This places additional burdens on SDN application developers as the design and implementation should now be explicitly tailored to be composition-aware. As we will show in §6, voting approaches can lead to inefficient solutions.

To summarize, existing composition approaches are output-driven, and produce resource-inefficient and/or unfair results. While recent work (e.g., [5]) improves upon naive composition approaches, it requires applications to be aware of all other applications on the network.

3 CHOPIN OVERVIEW

Chopin composes applications by creating a unified optimization problem from the application *intents*, instead of their outputs. Access to application intents enables composition with several attractive features:

- *Resource efficiency.* By creating a unified optimization, Chopin can simultaneously compute resource allocation for different traffic classes managed by different applications. Therefore, techniques that artificially constrain the pool of available solutions (e.g., static allocation or ordered optimization) are not used and do not negatively impact the solution.
- *Fairness.* Multiple applications necessitate multi-objective optimizations, moving away from a single notion of optimality. Fairness can be directly incorporated during the solution generation stage, as opposed to embedded after solutions have been computed.
- *Transparent composition.* The composition process is transparent to application developers, who write applications in the given intent framework. Intents are composed without additional developer effort.

Having explored the benefits of intent-driven composition, we outline the requirements for system design. Chopin should:

- Enable developers to specify application intents;
- Construct an integrated optimization (i.e., model resource usage, introduce fairness); and
- Produce the solution quickly enough to respond to network and traffic changes.

We detail our approach for each requirement below.

Intent specification. The goal of intent specification is to support different types of applications and model their requirements as an optimization problem. As observed in prior work [16, 21, 44], the *intents* of many network applications can be succinctly captured by specifying four key properties listed below:

- Types of traffic the application manages (i.e., traffic classes);
- Routing policy (a description of valid routing paths);
- Resource costs and constraints; and
- The objective function (e.g., minimize resource load, maximize flow).

Multiple types of conflicts may arise between application intents. We identify three groups of conflicts: (1) Policy conflicts, where applications want to apply different routing policy to the same traffic. We assume that policy conflicts between applications are resolved prior to running Chopin (e.g., with a tool like PGA [36]). Such resolution results in non-overlapping traffic classes, each with a unique routing policy. (2) Objective conflicts, where applications have conflicting global optimization goals; and (3) Resource conflicts, where applications assign different costs for a given traffic class and resource. Chopin focuses on resolution of resource and objective conflicts (described in detail in §4), with the help of a framework for intent specification.

Many prior efforts [16, 21, 44] provide an API and can support a variety of resource-management applications and their intents

(e.g., [18, 20, 29, 37, 42]). We use SOL [21] as a starting point because of our familiarity from prior work; our composition-centric challenges and insights apply to these other concurrent efforts as well. However, SOL and other efforts are designed to produce a solution for only one application at a time. Since our goal is to enable development of composition-*unaware* applications, we utilize this API, but design a new way to construct and solve the optimization.

Background on SOL and intent API. SOL's API presents a way to specify an assortment of resource-management objectives and constraints. By adopting network *paths* (as opposed to links) as the base unit for resource allocation, the API gains notable expressive power. More specifically, SOL provides an abstraction for expressing a wide range of traffic engineering, service chaining and NFV applications.

The developers make SOL API calls that specify resource costs for different traffic classes, define resource-management objectives, and control network elements (e.g., logically enable/disable links and nodes). For example, a traffic engineering application requires only three API calls: 1) define bandwidth as a resource; 2) route all traffic given a cost value per flow; and 3) minimize bandwidth load. SOL also supports arbitrary routing policies by filtering network paths according to a functional predicate. Resource loads are expressed using path-based constraints. The path-based design allows significant expressiveness, but introduces new challenges, namely efficient solving of the optimization. To keep the computation scalable despite the large (potentially exponential) number of paths in the network, SOL optimizes over a small subset of network paths chosen using heuristics.

Creating and solving an optimization. When constructing the optimization, Chopin leverages the mathematical representation of intents available in SOL, but introduces coordination and fairness among different applications. Given the application intents, Chopin resolves any resource-cost conflicts; i.e., if multiple applications try to manage the same traffic, then Chopin conservatively reconciles any discrepancies in how they measure the costs of that traffic by choosing the worst-case resource cost (formal definition provided in §4.1). During construction of the optimization, the network operator specifies the desired fairness metric for the applications' objectives and any additional resource constraints that apply to all applications.

Chopin splits the computation into two stages: offline path selection and online optimization. In the offline stage, valid routing paths are generated, and a subset of them are selected for use in the online optimization. Computationally intensive offline pre-processing keeps the online optimization scalable, which helps adjust to shifting traffic demands, if needed. Unlike the offline/online split utilized in SOL, Chopin's offline path selection is *coordinated*, which means it is tailored to a given set of applications. To see why coordination during path selection is necessary, consider the network in Fig. 4: two applications are required to choose two paths, and they pick the shortest available, as suggested by a previous path selection heuristic [21]. While sufficient for each application on its own, the total capacity of the paths is too low to carry traffic from both applications.

Similarly, changing network conditions adds greater complexity to the optimization, exacerbating the problems with offline path

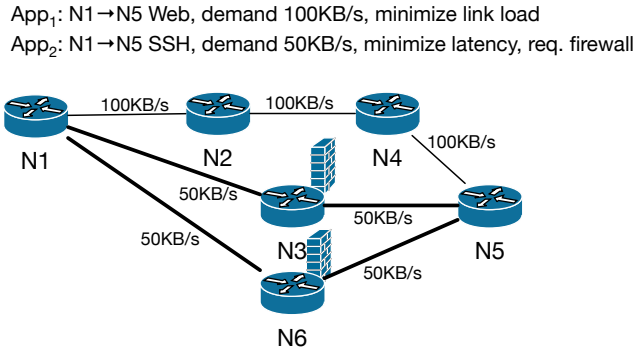

Figure 4: Naive composition using intent-based framework has no available solution. Both applications choose shortest paths (in bold), which are sufficient per application but lack capacity in a composition scenario.

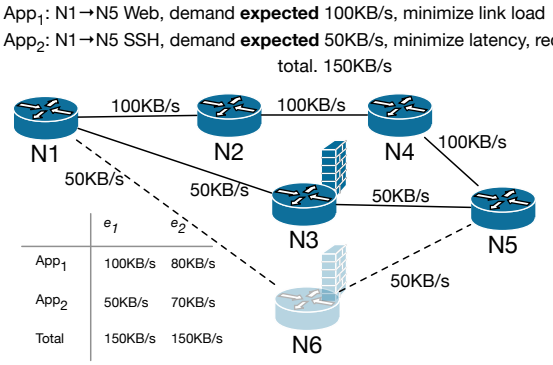

Figure 5: Traffic shift from time epoch e_1 to e_2 causes a policy violation if path N_1-N_6-N_5 is not chosen, despite the total volume of traffic remaining the same.

selection. Consider the network in Fig. 5, where a traffic volume shift occurs between time epochs e_1 and e_2. Despite the total volume of traffic being the same, App_2 cannot route traffic according to the policy as path N_1-N_2-N_4-N_5 does not conform to the policy and path N_1-N_6-N_5 was not selected during pre-processing. Multi-application path selection needs to account for such potential traffic shifts between applications to avoid infeasibility pitfalls.

End-to-end workflow. The full operational workflow is presented in Fig. 6. Application intents are specified using the provided API (step ❶). The operator collects the applications to be deployed and Chopin generates network paths that conform to the applications' routing policies. The operator then generates a collection of traffic matrices, one per *epoch* (step ❷). The temporal variability of the traffic matrix across epochs ensures the robustness of the solution and aids selection of a diverse set of paths. This "provisioning" traffic matrix can be generated from past observations or using synthetic models (e.g., [47]). The coordinated path selection (step ❸) selects a set of paths for each application, by composing the applications and choosing the paths that produce best results across the per-epoch traffic matrices. Paths are saved for later use in the online deployment phase. After pre-processing, the operator proceeds to deploy the applications (step ❹). Chopin constructs an

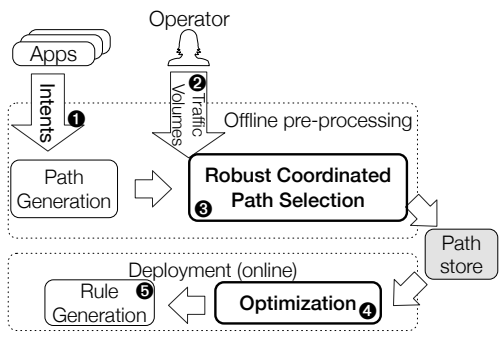

Figure 6: Chopin workflow: The operator collects application intents and traffic demands. Chopin performs robust path selection, and deploys the applications. (Chopin contributions are highlighted in bold.)

online optimization based on current (measured) network demands and applies the desired fairness metric. When running the online optimization, Chopin uses paths chosen during path selection (step ❸) as input to the optimization. The solution is converted to network flows, which are deployed via a network controller (step ❺) (e.g., [48]).

4 DETAILED DESIGN

In this section we focus on the steps that represent our primary technical innovations from Fig. 6: step ❹ (unified optimization, described in §4.2) and then step ❸ (coordinated path selection, described in §4.3). Unified optimization enables fair composition, while coordinated path selection enables a resource-efficient solution even in the presence of multiple applications and traffic variability.

4.1 Preliminaries

Composition is performed by combining application intents. First, applications' traffic classes and policies are used to generate valid network paths. Then, traffic is routed along these paths, consuming network resources as specified by applications' resource costs. Resource consumption, in turn, is reflected in the applications' objective functions. Precise notation follows below.

Traffic classes. A *traffic class* c is a subset of all traffic arriving at a designated ingress node $c.\mathsf{in}$, exiting at a designated egress node $c.\mathsf{out}$, and matching the specification $c.\mathsf{flowspec}$ (e.g., specified by IP 5-tuple). Each class c also has an associated volume estimate in number of flows per *time epoch* e, denoted $c.\mathsf{vol}[e]$. We assume that traffic classes do not overlap, i.e., if $\mathbb{C}$ is the set of all traffic classes, then for any $c_1, c_2 \in \mathbb{C}$, $c_1 \cap c_2 = \emptyset$. (Non-overlapping classes can be ensured by simply decomposing traffic into sufficiently fine-granted classes, e.g., [36].) A *traffic matrix* TM_e for epoch e has the value at location $TM_e[in, out]$:

$$\sum_{\substack{c \in \mathbb{C}\,:\, c.\mathsf{in} = in \\ \wedge\, c.\mathsf{out} = out}} c.\mathsf{vol}[e]$$

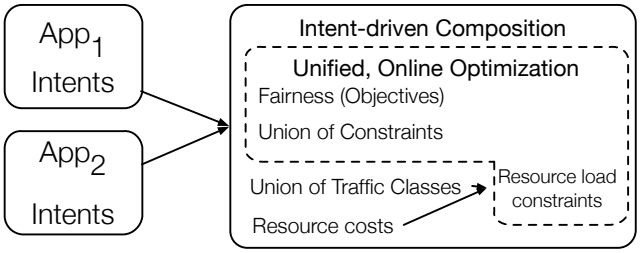

Figure 7: Conceptual composition of two applications: Unified optimization provides resource efficiency. Fairness is applied to the application objectives. Traffic classes and resource costs are used to compute resource load constraints.

Applications. For our purposes, an application *App* is specified as a set of traffic classes $App.\mathsf{classes} \subseteq \mathbb{C}$ that it manages; a set of permissible paths $App.\mathsf{paths}[in, out]$ for carrying traffic classes $c \in App.\mathsf{classes}$ such that $c.\mathsf{in} = in$ and $c.\mathsf{out} = out$; an average per-flow amount $App.\mathsf{cost}[r]$ of resource r consumed by traffic associated with this application; an objective function $App.\mathsf{obj}$ specified in terms of those resource costs and the network topology (e.g., maximizing flow, minimizing resource load); and various constraints that characterize allowable allocations of the traffic in $App.\mathsf{classes}$ to the network. The set $App.\mathsf{paths}[in, out]$ is generated in step ❶ of Fig. 6 to contain the paths that satisfy a predicate specified by the app developer (as a function that Chopin evaluates on candidate paths, as in SOL). Each node N on path $p \in App.\mathsf{paths}[in, out]$ has a fixed resource-r *capacity* $N.\mathsf{cap}[r]$, specified in the same units as $App.\mathsf{cost}[r]$. Similarly, each link L on path $p \in App.\mathsf{paths}[in, out]$ has a fixed resource-r *capacity* $L.\mathsf{cap}[r]$.

We define the per-flow cost for resource r associated with traffic class c as the maximum cost specified across all applications for a given traffic class:

$$c.\mathsf{cost}[r] = \max_{App:\ c \in App.\mathsf{classes}} App.\mathsf{cost}[r]$$

This ensures a conservative cost estimation across applications that manage the same traffic class c.
The allowable paths for c are

$$c.\mathsf{paths} = \bigcap_{App:\ c \in App.\mathsf{classes}} App.\mathsf{paths}[c.\mathsf{in}, c.\mathsf{out}]$$

which we assume to be nonempty.

4.2 Online, Unified Optimization

Chopin achieves fair and resource-efficient composition by creating a single unified optimization, which allows simultaneous optimization over multiple criteria (e.g., balancing middlebox load and link load simultaneously). Fig. 7 provides a conceptual view for the composition process: a single online optimization is constructed from the application's resource-management intents. A fairness measure is applied to the applications' objective functions. Application-specific constraints are combined unmodified, while resource load constraints are computed from the traffic classes and resource costs provided by each application. Fig. 8 describes the mathematical underpinnings of the optimization. We emphasize

maximize

$$Obj = \sum_i w_i \times App_i.\text{obj}[\mathbb{E}] \quad (1)$$

subject to, for all $e \in \mathbb{E}$,

$$\cdots$$

$$NLoad^c_N[r, e] = \sum_{\substack{p \in c.\text{paths}: \\ N \in p}} x_{c,p,e} \frac{c.\text{cost}[r] \times c.\text{vol}[e]}{N.\text{cap}[r]} \quad (2)$$

$$0 \le x_{c,p,e} \le 1 \quad (3)$$

$$NLoad_N[r, e] = \sum_{c \in \mathbb{C}} NLoad^c_N[r, e] \quad (4)$$

$$NLoad[r, e] = \max_N NLoad_N[r, e] \quad (5)$$

$$NLoad[r, e] \le NLimit[r] \quad (6)$$

$$\cdots$$

$$\sum_{c \in \mathbb{C}} \sum_{p \in c.\text{paths}} b_{c,p} \le |\mathbb{C}| \times NumPaths \quad (7)$$

$$0 \le x_{c,p,e} \le b_{c,p} \quad (8)$$

$$b_{c,p} \in \{0, 1\} \quad (9)$$

Figure 8: Core components of the linear programming formulation of the unified optimization. An example resource load computation is described in Eqn. 2–Eqn. 6, where $\mathbb{E}$ is a singleton set (containing the current epoch index) in the online optimization and $\mathbb{E} = \{1, \ldots, NumEpochs\}$ in the offline path selection. Offline path selection also adds Eqn. 7–Eqn. 9.

that for this subsection, the epoch index e is a constant, and $\mathbb{E}$ denotes the singleton set $\mathbb{E} = \{e\}$. This is relaxed in §4.3.

Resource load and objectives. Resources represent any capacity-bounded property of links or nodes (e.g., bandwidth, TCAM space, CPU speed). To standardize resource consumption, all resource-r loads and objectives are normalized to a standard range of $[0, 1]$, using the resource-r capacity $N.\text{cap}[r]$ per node N and $L.\text{cap}[r]$ per link L. Load on resources is expressed per traffic class using the network paths available for that traffic class. For example, for resources relevant to nodes, we define the resource-r load $NLoad^c_N[r, e]$ induced by traffic class c on node N during epoch e by Eqn. 2, where $N \in p$ represents that node N lies on path p and where $x_{c,p,e}$ is a variable representing the fraction of flows of traffic class c routed on path p during epoch e. Then, we define the load on a resource r at node N as the sum of loads imposed by all traffic classes (Eqn. 4), and require these loads for all nodes to be at most $NLimit[r]$ (Eqn. 6), an operator-specified constant. Links are treated similarly.

Objectives represent a global resource-management intent (e.g,. maximizing spare capacity of links). Chopin supports a number of predefined objective functions to maximize. Note that because objectives are normalized, any min optimization can be converted to a max optimization by using $1 - App.\text{obj}$ as the new $App.\text{obj}$. For example, a maximization objective that minimizes the load on node resource r is

$$App.\text{obj}[\mathbb{E}] = 1 - \max_N \sum_{c \in App.\text{classes}} NLoad^c_N[r, e] \quad (10)$$

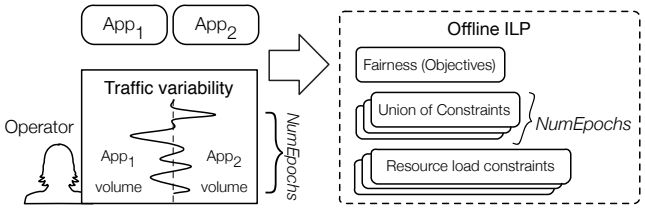

Figure 9: Offline coordinated path selection provides robustness by constructing a unified optimization with sets of constraints for each epoch, thus ensuring resource efficiency and fairness in each epoch.

The combined optimization objective for the composed applications is computed according to a specified fairness metric (see below) and maximized, subject to the constraints of all of the applications.

Fairness metrics. To ensure that no single application dominates the solution, we need some notion of fairness. There are two natural ways of specifying fairness: (1) as some measure of the individual objective functions or (2) as a measure of the consumption of various network resources such as link bandwidth. We opt for applying the fairness metrics to the applications' objectives, for two reasons. First, the objectives allow for a unified way of enforcing fairness across applications. Second, mandating fair *use* of resources can result in non-linear equations with respect to $x_{c,p,e}$ variables, thus sacrificing many of the scalability benefits of linear programming optimizations.

Chopin's design is general and can support many commonly used fairness metrics defined over the individual application objectives.

- Most linear functions can be directly incorporated into the optimization: For example, weighted combination of objective functions results in a utilitarian solution, shown in Eqn. 1, where w_i is a weight assigned to each application. In our case, we find it makes sense to automatically assign weights proportional to the volume of traffic that the application is routing.
- Another common metric is the Rawlsian difference principle for maximizing the minimum objective [15]:

$$\texttt{maximize } Obj = \min_i App_i.\text{obj}[\mathbb{E}] \quad (11)$$

- Chopin can also (approximately) support proportional fairness [25] defined as:[2]

$$\texttt{maximize } Obj = \sum_i \log App_i.\text{obj}[\mathbb{E}] \quad (12)$$

- At a slightly higher computational cost (due to their quadratic nature), the relative mean deviation and variance functions [3] can be supported.

4.3 Offline, Coordinated Path Selection

A key innovation in Chopin is selecting paths in an offline phase to ensure that the available paths are (i) rich enough to offer adequate capacity to support all applications but also (ii) few enough to permit the online, unified optimization above to be solved fast enough to ensure responsiveness on network timescales. For this

[2]Since a log function cannot be directly incorporated into a linear program, Chopin implements a piece-wise linear approximation [8].

purpose, we leverage the unified optimization described in §4.2 to construct a path selection integer linear program (ILP). The resulting ILP chooses paths capable of achieving resource efficiency under traffic variations (as specified by the operator) by creating a set of constraints per traffic matrix epoch (overview shown in Fig. 9).

Formally, this is achieved by augmenting the unified optimization with additional constraints, shown as Eqn. 7–Eqn. 9 in Fig. 8, and broadening the optimization to maximize per-application objectives *across epochs* $\mathbb{E} = \{1, \ldots, NumEpochs\}$, i.e., where

$$App.\text{obj}[\mathbb{E}] = \frac{1}{NumEpochs} \sum_{e \in \mathbb{E}} App.\text{obj}[\{e\}]$$

(see Eqn. 1). Eqn. 7 specifies a global limit on the number of paths used. The cap is computed using a baseline of *NumPaths* paths per traffic class, although the final number of paths per traffic class can deviate from *NumPaths* to achieve better results. Eqn. 8 ensures that only chosen paths are allowed to carry flow.

Tractability. The resulting ILP presents tradeoffs between resource efficiency and scalability. A larger number of epochs provides a solution more accommodating to traffic variations and thus typically yielding better resource efficiency, at the cost of runtime and memory needed to perform offline path selection. Similarly, an increase in the network size and number of paths (and thus number of binary $b_{c,p}$ variables) renders computing a true-optimal solution intractable. To address these challenges, we propose two scalability improvements, clustering and relaxed path search, described below.

Clustering speedup. To reduce the problem size, we must reduce the number of traffic matrices (epochs), yet do so without significantly reducing the variability they represent. We exploit the fact that network traffic volumes (and their synthetic models) exhibit patterns that we can preserve if we employ a *clustering* technique. Hence, we cluster traffic classes across epochs based on their volumes. Specifically, if $\langle c_1, c_2, \ldots \rangle$ is a fixed ordering of the traffic classes, then we cluster the vectors $\{\langle c_1.\text{vol}[e], c_2.\text{vol}[e], c_3.\text{vol}[e], \ldots \rangle\}_{e \in \mathbb{E}}$. We have experimented with a number of clustering techniques and find that k-means clustering [17] is the most scalable approach. However, the output of k-means clustering is a set of centroids that represent the *average* traffic volumes for each cluster, causing path selection to not consider the worst-case scenario. This is acceptable in many cases, however for added robustness, alternative solutions are available. First, we can apply a scaling factor to the computed centroid, which allows over-provisioning. Alternatively, we can use a different clustering algorithm. For example, we also implement a hierarchical clustering algorithm with Ward's linkage [49]. Ward's algorithm allows us to group traffic volumes based on their similarity into clusters, leaving us with the possibility of applying a custom reduction function (e.g., max). That is, given *NumClusters* groups of traffic classes, we can use the worst-case volumes from each group instead of the average volumes.

Relaxed path search. Unfortunately, even with the clustering described above, solving the ILP remains a challenge. Increases in the number of paths (due to topology size or number of traffic classes) quickly makes computing a solution impractical due to time and memory consumption. Hence, we introduce an iterative path search approach that does not require solving an ILP.

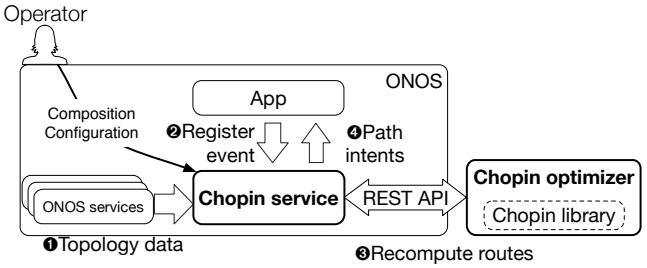

Figure 10: Integration between Chopin and ONOS. ONOS applications register with the Chopin service which triggers a computation using the Chopin optimizer. The solution is converted to path intents and returned to the application.

After the traffic classes have been clustered, relaxed path search starts by attempting to compute the solution with a small number of paths per traffic class (e.g., *NumPaths* = 5). Each iteration of the search will increase the *NumPaths* value until the resulting objective value does not differ significantly from the objective obtained in the previous iteration (e.g., $\Delta \leq \epsilon$, where $\epsilon = 0.05$). This indicates that there is no benefit to adding more paths. Finally, the union of all flow-carrying paths are saved for the online optimization stage. A path p is considered flow-carrying if the flow fraction $x_{c,p,e} > 0$ for at least one epoch e.

The intuition behind iterative search is that multi-path routing is of limited value for topologies with sufficiently many traffic classes. This has been analytically shown for routing problems with a single resource and objective function [1, 30] and we observe similar behavior empirically for multi-application, multi-epoch optimizations. Therefore, we exploit the diminishing returns of adding more paths to the optimization by iteratively increasing the number of available paths per traffic class until the objective value no longer improves. Lastly, the union of flow-carrying paths across epochs is taken, and provides sufficient degrees of freedom for the online optimization.

Search heuristics. The success of iterative path search hinges on the heuristic logic responsible for choosing paths to be added in the next iteration. A natural approach is to adopt a greedy heuristic (e.g., based on path length or edge-disjointness). However, we find that this heuristic does not perform as expected in the multi-application scenario. Our view is that multi-objective optimization necessitates a multi-criteria heuristic. Therefore, our heuristic scores the paths based on length and "resource-richness". That is, it favors shorter paths that maximally augment the resources available to the application (based on the applications' objective functions). This biases the selection towards lower latency and link consumption (as bandwidth is a shared resource among all applications), yet still provides sufficient freedom to load balance the applications' resources of interest.

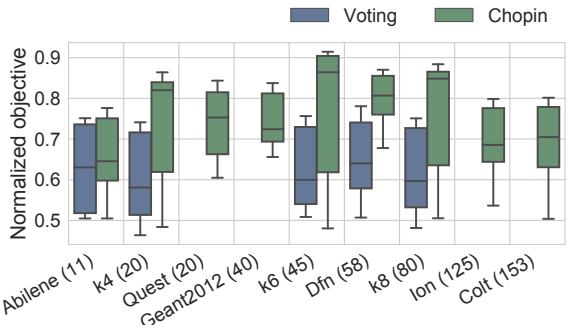

Figure 11: Optimality comparison between Chopin and Athens-like voting framework.

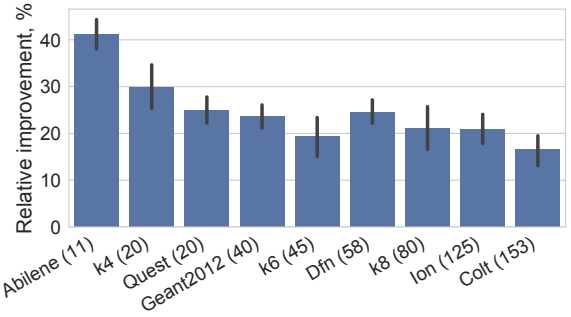

Figure 12: Relative improvement in objective function when using Chopin as opposed to static allocation methods with shortest paths per application. Averaged across 100 epochs.

5 IMPLEMENTATION

Chopin Library. Our library for expressing and solving optimizations is implemented in Python and can be used for composing optimizations and generating solutions as described in §4. The library requires a linear programming solver; our prototype uses Gurobi [14]. The source code for the library is available on Github [19].

Chopin Optimizer. Atop the library we built the Chopin optimizer, a standalone component capable of receiving composition requests from an SDN controller (or other applications). The optimizer exposes an HTTP REST API, allowing the integrations to be built in a multitude of languages and runtimes.

Integration with ONOS. For our prototype, we implemented a Chopin service in the ONOS controller. Fig. 10 depicts an architectural view of the ONOS component and its interaction with the Chopin optimizer. The Chopin service is deployed inside ONOS and receives network data (e.g., states of devices and links) from other ONOS services (step ❶). A newly deployed application registers with the service and provides its optimization intents to the Chopin service (step ❷). The register event starts the re-computation process, which utilizes the REST API to communicate with the optimizer, and requests the composition of all applications registered up to this point (step ❸). The Chopin service parses the solution received from the optimizer, generates appropriate ONOS path intents, and returns them to the application(s) (step ❹). The service also allows the administrator to specify global network constraints that will act across applications.

ONOS controller also allows the Chopin service to trigger re-computation of traffic allocation based on observed traffic patterns. That is, if network monitoring suggests a significant deviation from the last traffic matrix used for optimization, a new optimization may be triggered.

6 EVALUATION

In this section, we evaluate Chopin using trace-driven simulations. We describe the following results:

- Resource-efficiency improvements over static allocation and voting approaches (Fig. 11, Fig. 12);
- Impact of different fairness metrics on the solution (Fig. 13);
- Runtime improvements in scalability of path selection due to clustering and relaxed path selection (Fig. 14) and low impact of traffic estimation errors (Fig. 16);
- End-to-end validation using the ONOS controller (Fig. 17).

Setup. We chose topologies of various sizes from the Topology-Zoo dataset [26]; when indicating a topology, we generally include the number of nodes in the topology in parentheses, e.g., "Abilene (11)" for the 11-node Abilene topology. We also constructed FatTree topologies of various sizes [2]. We refer to these as "kX" where X denotes the arity of the FatTree, as defined in prior work. We synthetically generated traffic matrices using a modulated gravity model [39] and introduced a temporal variation between applications' traffic volumes using a Dirichlet distribution [23] across 100 epochs. We chose the Dirichlet distribution because it generates a worst-case variability in traffic shifts between applications (e.g., 0/100% to 100/0% split between two applications), while maintaining fixed traffic volume across epochs. We also performed tests with other variability models (e.g., [47]) and observed similar results.

Unless otherwise specified, we used two canonical applications—a traffic engineering application that minimizes link load and a service chaining application that minimizes middlebox load. The applications had no overlapping traffic classes and were composed using a weighted fairness metric, with the weights proportional to the volume of traffic that belonged to the application. The service chaining application required a chain of two middleboxes — a firewall and an intrusion detection system. Times below refer to computation on computers with 2.4GHz cores and 128GB of RAM, except end-to-end benchmarks, where we used Mininet [31] to emulate the topologies.

6.1 Resource Efficiency and Fairness

We use trace-driven simulations to evaluate the benefits of using Chopin for resource efficiency, fairness, and responsiveness by comparing it to output-driven approaches. We also evaluate the potential benefits of using Chopin's coordinated path selection approach compared to prior work [21].

Comparison to output-driven approaches. We compare Chopin to two output-driven approaches: naive static allocation and Athens [5]. We chose Athens because it is arguably the closest practical work in the resource-management composition space. We created a simulator that implements the Athens voting protocols and modified our applications to be aware of all other applications present. We considered a set of paths and their flow allocations to

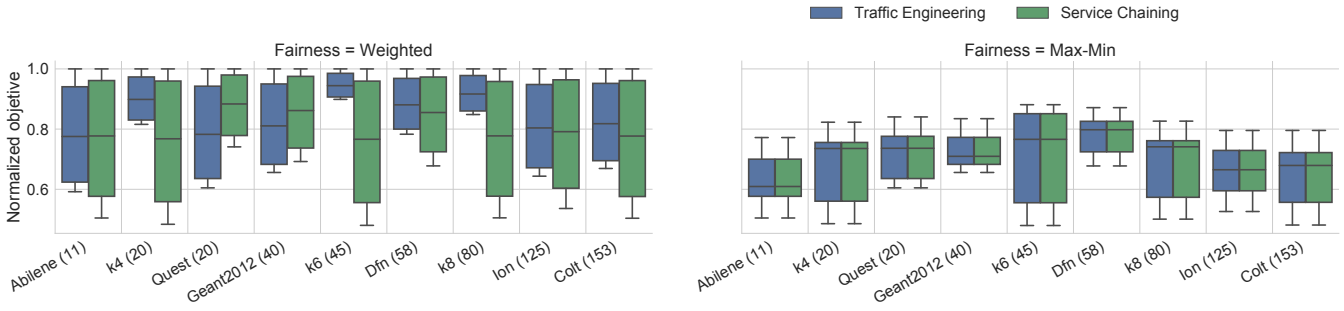

Figure 13: Impact of chosen fairness metric on the objective function of each application

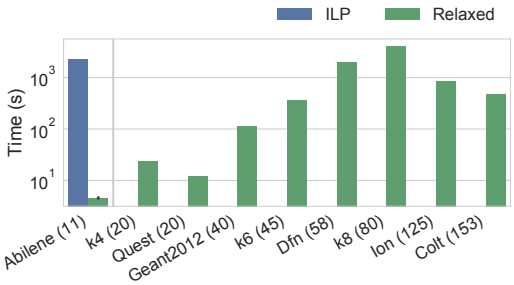

Figure 14: Runtime comparison of the optimal ILP path selection and relaxed selection. Relaxed paths selection is orders of magnitude faster than optimal ILP path selection.

be a "proposal" that is submitted for voting. Each application generated a selfish proposal (i.e., best for its own objective function) and submitted it. Votes on the proposals were cast inverse-proportional to the deterioration of the application's objective function. Fig. 11 shows the comparison in the *global* objective function between the voting approach and Chopin (higher objective is better). Each box plot in the figure represents a composition strategy executed 100 times (i.e., across 100 epochs), with boxes covering objective values between the 25th and 75th percentiles and whiskers extending to min and max values. Chopin outperforms the voting approach by as much as 60%, and in some cases voting failed to converge entirely (e.g., Geant and Quest topologies).

Benefits of coordinated path selection. We compared solutions obtained using shortest paths and solutions produced by Chopin. Fig. 12 shows the relative improvement over the baseline objective computed using shortest paths. Bars represent the improvement averaged across 100 epochs, with error bars indicating the standard deviation. For all topologies, Chopin produces a more resource-efficient solution than the naive shortest-path selection strategy.

Impact of fairness metrics on objectives. To explore the effect of composition fairness on the objective functions of individual applications, we composed two applications using two different fairness metrics: weighted fairness (i.e., weighted proportional to the application traffic volume), and max-min fairness (recall §4.2). Fig. 13 shows objectives of two applications across different topologies and fairness metrics. Max-min fairness is arguably the "most fair", ensuring equal objectives but not achieving the best load balancing for either of the applications. In contrast, weighted fairness maximizes the global objective, but does so at a cost of application inequality (e.g., favoring the traffic engineering on the k6 topology). This result highlights that Chopin is flexible and gives the operator ability to customize the solution, be it in favor of overall network resource efficiency or fairness.

6.2 Runtime benchmarks

Path selection benchmarks. To show runtime benefits of Chopin's path selection, we compare the optimal ILP described in §4 and relaxed path selection. The ILP is difficult to compute for all but the smallest topology. Not using the ILP allows orders of magnitude faster offline path selection (Fig. 14).

Online optimization benchmarks. Finally, we demonstrate that Chopin's online component is also scalable. We composed different combinations of applications (traffic engineering, service chaining, latency minimization), up to 5 total, using Chopin and static allocation with SOL single-application optimization framework. Both setups used the same number of paths, 5 per traffic class. Fig. 15 depicts the mean time (across 100 epochs) to construct and solve the unified optimization (in case of Chopin) or series of optimizations (in case of static allocation) across a number of topologies, for different numbers of applications. The runtimes are similar, and follow the same patterns across topologies and numbers of applications. However, Chopin achieves better optimality (recall Fig. 12).

Robustness to estimation error. We also evaluated the impact of traffic estimation errors on traffic matrix clustering and path selection. To do so, we generated traffic matrices for different topologies and used them to perform paths selection as described in §4. We then introduced noise to the traffic matrices by changing the volumes of each traffic class across different epochs. The noise was relative to the mean traffic volume of each traffic class and was sampled from a truncated normal distribution (with $\mu = 0$ and $\sigma = 0.2$). Fig. 16 depicts the relative error of using Chopin's pre-selected paths compared to the optimal solution (using all paths for each epoch and perfect knowledge of the traffic matrix) for topologies where the optimal solution could be computed in reasonable time. The box plots show the median, 1st and 3rd quartiles, with whiskers extending to min and max values. The result indicates that in over

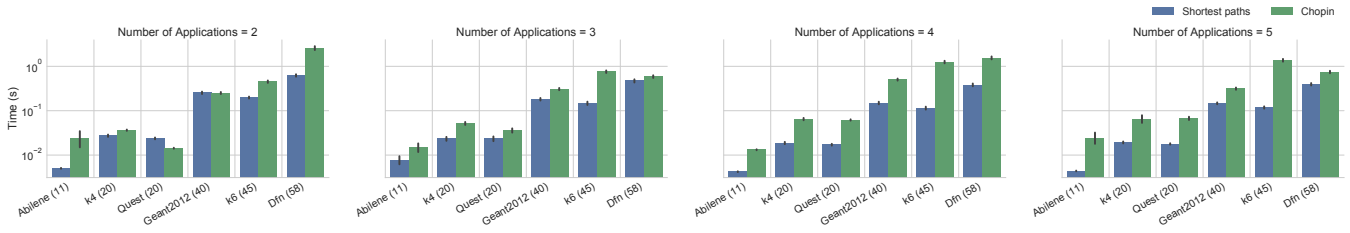

Figure 15: Mean time to execute a single-epoch optimization. Chopin scales similarly to using SOL in conjunction with static allocation across topologies and numbers of applications.

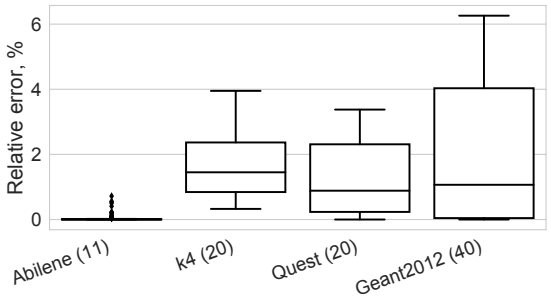

Figure 16: Impact of traffic estimation error on solution quality

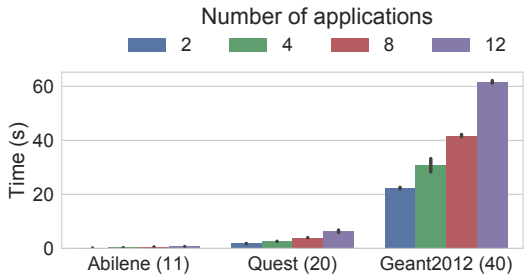

Figure 17: Time to deploy multiple applications as a function of number of applications. Includes time for online optimization and computing and installing ONOS path intents.

half the cases the error was quite small, $\leq 2\%$, and ranged only up to around 6% in the worst case.

6.3 End-to-end validation

We set up different topologies using the Mininet emulator and ONOS controller. We deployed up to 12 applications on each topology. Specifically, we use a combination of traffic engineering and service chaining applications with partially overlapping traffic classes. Fig. 17 shows the worst-case time to deploy the applications, meaning each new application triggered a full re-computation for all traffic classes for all applications. Even the worst-case deployment (i.e., configuring the network from scratch) maintains network time-scale responsiveness.

7 DISCUSSION

Expressible applications. At this time, Chopin's focus is geared towards resource-management applications. The intent API provided by SOL [21] is sufficient to support a multitude of such applications, as it allows a variety of resource-consumption models (e.g., per link, node, middlebox, and even path for TCAM constraints). Applications that utilize an ILP formulation (e.g. [18, 37]) are also supported.

Interaction with the SDN ecosystem. Applications that do not perform resource management can easily co-exist with applications written in Chopin without interference. Resource management applications outside of the Chopin framework will not get the coordination benefits, while Chopin applications will coordinate and perform as intended if the traffic estimation is reasonable.

Discrete ordering of objectives. In some cases the operator might desire an "if-then" ordering of applications. For example, perform load-balancing only after the QoS latency requirements have been met. Our design philosophy is to avoid such discretization, in favor of resource-efficiency. However, application ordering can be modeled by converting objective functions to hard constraints in the optimization. For example, instead of minimizing latency, a constraint is added to enforce a maximum-allowable latency.

Choosing fairness. We intentionally allow flexibility for the operator to choose a fairness metric that best suits her needs. As we showed in §6, the effects of choosing different fairness metrics are noticeable. However, as a starting point we suggest weighted fairness, which will maximize network's resource efficiency as a whole.

Unpredictable traffic changes. As we discussed in §6, normal deviations from expected traffic matrices have a low impact on resource efficiency. However, if the observed traffic pattern differs significantly from the estimations used in the offline pre-processing stage, Chopin will still attempt to find an available solution, at the cost of a larger resource-efficiency gap.

Network reconfiguration. When traffic changes or new applications trigger online re-optimization, a new solution is generated from scratch. To avoid disrupting existing network flows, Chopin can update only the traffic classes where network flows have changed as a result of the optimization. To reduce such churn even further, existing techniques can be employed for minimizing both optimization [21] and flow rule [24] differences.

Limitations Chopin's algorithms require further augmentation to perform well on highly unstable networks (e.g., networks with frequent intermittent link failures). Since a link failure can affect

one or more paths, operators using Chopin would need to guarantee sufficient over-provisioning of alternative paths or pre-compute fast fail-over solutions, if the set of expected failures is known.

8 RELATED WORK

SDN application composition. The closest work to ours that also focuses on composition of resource-management applications is Corybantic [4] and its successor Athens [5]. Unlike Chopin, their approach requires modified, composition-aware applications. Works such as Covisor [22], Flowvisor [43] and Frenetic [12, 32] focus on OpenFlow rule composition and packet-level policy chain composition. While important in the space of composition, these prior works do not address resource management. PGA [36], Statesman [46], and PANE [11] focus on resolving policy conflicts between applications and enforcing network invariants. Policy composition is orthogonal to resource management, and can be viewed as a step preceding resource management; e.g., the composed policies produced by PGA are used to generate sets of valid paths prior to selection and optimization. Finally, we also note the industry-proposed projects by ONOS [35] and OpenDaylight [33], which appear to be policy-focused as well.

Optimization frameworks. SOL [21], Merlin [44], and DEFO [16] are precursors to our work, and provide the necessary start for intent-driven composition. However, their designs are focused on a single-application use case only. We showed that naively extending these frameworks for multi-application use presents new challenges, namely lack of fairness and resource-efficiency. Our work addresses these limitations, proposes a design that separates application development and deployment concerns, and produces fair, resource-efficient solutions.

Robust optimization. Robust optimization [7] develops ways of "protecting" the optimizations against uncertain data. In networking, this takes on the form of network design validation against failures [9] or (semi-)oblivious routing [6, 27, 28]. Such techniques, while possibly conservative and computationally intensive, can be complementary to our work and can be incorporated into the internals of Chopin. We leave this for future work.

Multi-objective optimization. Extensive literature exists on multi-objective optimization, covering a number of techniques [10, 45]. The focus of aforementioned works, however, is not on network optimization and often requires manual intervention. Our work is specific to automatic composition of networking applications.

9 CONCLUSIONS

A growing number of SDN resource-management applications necessitates a new way of composing them that guarantees fair and resource-efficient solutions. Our goal is to provide a system capable of achieving such composition while abstracting away low-level composition details from application developers and network operators. In this paper, we presented Chopin — an intent-driven composition framework for SDN applications. It builds upon the concepts of intent-driven network optimization to provide fair and resource-efficient composition. We showed that Chopin achieves significantly better resource-efficiency than previous output-driven approaches, can be integrated with modern SDN controllers, and outperforms uncoordinated approaches to composition.

Acknowledgements

This work was supported in part by NSF grants 1535917 and 1536002.

REFERENCES

[1] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. 1993. *Network flows: theory, algorithms, and applications.* Prentice hall.
[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. 63–74.
[3] Anthony B Atkinson. 1970. On the measurement of inequality. *Journal of economic theory* 2, 3 (1970), 244–263.
[4] Alvin AuYoung, Sujata Banerjee, Jeongkeun Lee, Jeffrey C Mogul, Jayaram Mudigonda, Lucian Popa, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *ACM HotNets*.
[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 *ACM CoNEXT*. ACM, 391–402.
[6] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Racke. 2003. Optimal oblivious routing in polynomial time. In *ACM Symposium on Theory of Computing*. ACM, 383–388.
[7] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. 2009. *Robust optimization.* Princeton University Press.
[8] Eduardo Camponogara and Luiz Fernando Nazari. 2015. Models and Algorithms for Optimal Piecewise-Linear Function Approximation. *Mathematical Problems in Engineering* (2015).
[9] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. 2017. Robust Validation of Network Designs under Uncertain Demands and Failures. In *USENIX Symposium on Networked Systems Design and Implementation*.
[10] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. 2016. Multi-objective optimization. In *Decision Sciences: Theory and Practice*. CRC Press, 145–184.
[11] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. In *ACM SIGCOMM*.
[12] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, Vol. 46. 279–291.
[13] Albert Greenberg. 2015. SDN for the cloud, Keynote. In *ACM SIGCOMM*.
[14] gurobi [n. d.]. Gurobi. http://www.gurobi.com/. ([n. d.]).
[15] Peter J Hammond. 1976. Equity, Arrow's conditions, and Rawls' difference principle. *Econometrica: Journal of the Econometric Society* (1976), 793–804.
[16] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. 2015. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *ACM SIGCOMM*.
[17] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108.
[18] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. ElasticTree: Saving Energy in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation*. 19–21.
[19] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. 2018. Chopin source code repository. https://github.com/progwriter/sol/. (2018).
[20] V. Heorhiadi, S. K. Fayaz, M. K. Reiter, and V. Sekar. 2014. SNIPS: A software-defined approach for scaling intrusion prevention systems via offloading. In *10th International Conference on Information Systems Security*.
[21] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. 2016. Simplifying software-defined network optimization using SOL. In *USENIX Symposium on Networked Systems Design and Implementation*. 223–237.
[22] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. Covisor: A compositional hypervisor for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation*. 87–101.
[23] Norman L Johnson, Samuel Kotz, and N Balakrishnan. 2002. *Continuous multivariate distributions, volume 1, models and applications*. Vol. 59. New York: John Wiley & Sons.
[24] N Kang, M Ghobadi, J Reumann, and A Shraer. 2015. Efficient Traffic Splitting on SDN Switches. In *ACM CoNEXT*.
[25] Frank P Kelly, Aman K Maulloo, and David KH Tan. 1998. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society* 49, 3 (1998), 237–252.

[26] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (October 2011), 1765 –1775. https://doi.org/10.1109/JSAC.2011.111002

[27] Murali Kodialam, TV Lakshman, and Sudipta Sengupta. 2011. Traffic-oblivious routing in the hose model. *IEEE/ACM Transactions on Networking* 19, 3 (2011), 774–787.

[28] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, and Robert Soulé. 2016. Kulfi: Robust traffic engineering using semi-oblivious routing. *arXiv preprint arXiv:1603.01203* (2016).

[29] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, Anja Feldmann, et al. 2014. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX Annual Technical Conference*.

[30] Xuan Liu, Sudhir Mohanraj, Michal Pioro, and Deep Medhi. 2014. Multipath Routing From a Traffic Engineering Perspective: How Beneficial is It?. In *IEEE International Conference on Network Protocols*. 143–154.

[31] mininet [n. d.]. Mininet. http://mininet.org/. ([n. d.]).

[32] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. Composing Software Defined Networks.. In *USENIX Symposium on Networked Systems Design and Implementation*, Vol. 13. 1–13.

[33] network-intent [n. d.]. Network Intent Composition. https://wiki.opendaylight.org/view/Project_Proposals:Network_Intent_Composition. ([n. d.]).

[34] Navid Nikaein, Eryk Schiller, Romain Favraud, Kostas Katsalis, Donatos Stavropoulos, Islam Alyafawi, Zhongliang Zhao, Torsten Braun, and Thanasis Korakis. 2015. Network store: Exploring slicing in future 5G Networks. In *International Workshop on Mobility in the Evolving Internet Architecture*. ACM, 8–13.

[35] onos-compose [n. d.]. Composition Mode. https://wiki.onosproject.org/display/ONOS/Composition+Mode. ([n. d.]).

[36] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM* 45, 4 (2015), 29–42.

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

[38] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular SDN Programming with Pyretic. *;login: Magazine* 38, 5 (2013), 128–134.

[39] Matthew Roughan. 2005. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review* 35 (2005), 4. https://doi.org/10.1145/1096536.1096551

[40] sdn-app-store 2017. SDN App Store. https://marketplace.saas.hpe.com/sdn. (January 2017).

[41] Kevin Shatzkamer. 2014. App store portal providing point-and-click deployment of third-party virtualized network functions. (April 4 2014). US Patent App. 14/245,193.

[42] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *ACM SIGCOMM*.

[43] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. 2009. *Flowvisor: A network virtualization layer*. Technical Report. OpenFlow Switch Consortium.

[44] Robert Soule, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *ACM CoNEXT*.

[45] Ralph E Steuer. 1986. *Multiple criteria optimization: theory, computation, and applications*. Wiley.

[46] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-state Management Service. In *ACM SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 563–574. https://doi.org/10.1145/2619239.2626298

[47] Paul Tune and Matthew Roughan. 2015. Spatiotemporal traffic matrix synthesis. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 579–592.

[48] R. Wang, D. Butnariu, and J. Rexford. 2011. Openflow-based server load balancing gone wild. In *Hot-ICE*.

[49] Joe H Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association* 58, 301 (1963), 236–244.