Robust Composition of Network and Security Applications using Software-Defined Networking

ABSTRACT

As SDN deployments mature, operators need to manage and compose multiple applications. Of particular interest are security and networking applications that measure as traffic engineering and service chaining, among others. In the absence of suitable composition abstractions, prior approaches result in suboptimal resource utilization, unfairness, and/or require applications to be rewritten to be aware of competing applications. To address these challenges, we present Chopin – a composition framework for SDN applications. By adopting a high-level, semantically rich framework for expressing applications, Chopin can automatically compose applications without needing their developers to rewrite the applications to be composition-aware. In designing Chopin, we address three dimensions of robustness w.r.t. application composition: resource efficiency, responsiveness, and fairness. By constructing a unified optimization, Chopin achieves better resource efficiency and fairness than previous “black-box” approaches. Chopin also utilizes a novel offline path-selection step to enable rapid online composition to be responsive to traffic variations. We implement Chopin, integrate it with the ONOS controller, and demonstrate that it substantially improves resource efficiency, responsiveness, and/or fairness over other approaches to composition.

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 operating large datacenters have repeatedly reported deploying multiple, specialized applications on their network [42, 12]. SDN needs the ability to compose applications and their functionality on a single network, considering the growth of application diversity and recent push for SDN “app stores” [36, 30, 37].

Such composition presents new challenges in the deployment of multiple applications. While policy composition has been studied extensively and many solutions are available [18, 34, 32], optimal resource management remains a hard problem due to a wide range of applications and their demands (e.g., load balancing, power saving, service chaining, intrusion detection, etc.) [33, 26, 15].

Today, most such resource management applications are expressed as standalone applications written in low-level optimization tools or using custom solvers. As such, these optimizations assume full control of the network and are constructed without regard for other applications, their resources, or traffic demands. Furthermore, optimizations (as presented to the solver) retain little to no semantic information about the network, resources, and intent of the optimization. Since composing optimizations at this level is impractical, previous work has avoided direct composition by taking “black box” approaches to composition: by partitioning resources between applications via topology virtualization [39, 24], or by requiring optimizations to be aware of other applications [3, 4]. The former is prone to producing suboptimal results, while the latter adds to developer burden, especially given the diverse set of applications imminent in the SDN ecosystem [37, 36].

A new trend of network optimization frameworks [40, 17] offers a promising alternative for SDN resource management applications. Such frameworks have access to an application’s high-level optimization context, allowing for a “white box” composition approach. This can ease the developer burden by abstracting away low-level composition details, and potentially offer better solutions than naive partitioning.
Unfortunately, the designs of such frameworks are not robust on three key dimensions discussed below, precluding them from being used as-is for composition:

- **Fairness:** Multiple applications lead to multi-objective optimizations i.e., moving away from a single notion of optimality. Hence, the framework must fairly balance multiple optimization criteria.
- **Resource efficiency:** multiple applications introduce variability into the network, since one application’s routing decisions can impact the decision-making process of co-existing applications by modifying network state. Optimizations, however, are sensitive to such data uncertainty, possibly causing the solution quality to degrade [6].
- **Responsiveness:** When resource efficiency suffers from traffic dynamics and competing applications, traffic allocations must be recomputed to offset these inefficiencies. Doing so, however, requires the ability to recompute and redeploy quickly, so as to be responsive at the timescales needed for the applications.

To this end, we propose Chopin (Fig. 1), a framework that enables robust composition of resource-management SDN applications with several attractive features. First, Chopin maintains single-application APIs exposed to the developers, keeping the development process agnostic to other applications. Second, it supports multiple composition modes based on different fairness metrics (e.g., [14, 20, 2]), providing flexibility for the operator. Third, it produces near-optimal composition results in short timescales, to respond to traffic variations.

To achieve these features, Chopin borrows from high-level frameworks for implementing network optimization applications, specifically building on the path-based optimization abstraction proposed in SOL [17]. To ensure that the resulting solution is optimal and fair, Chopin composes multiple applications into a unified optimization. To gain robustness while maintaining responsiveness to traffic changes, Chopin introduces an offline coordinated path selection, a step that prunes the set of available paths for scalability, while maintaining efficiency in the online optimization. We further improve the tractability of coordinated path selection using traffic matrix clustering and simulated annealing.

We have implemented a Chopin prototype using Python and a Chopin service in Java for the ONOS controller. We show that using the unified optimization and coordinated path selection Chopin achieves better optimality than naive approaches by as much as 10%. Chopin also outperforms black-box composition based on voting mechanisms in optimality by a factor of 2 and runtime by as much as an order of magnitude. Finally, we analyze Chopin’s scalability improvements by showing that with traffic matrix clustering and simulated annealing we achieve an order of magnitude speedup while sacrificing ≈ 1% in optimality.

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

- We empirically demonstrate the limitations of composing applications using black-box approaches;
- We introduce white-box composition as an alternative, leveraging the semantically rich representations of optimization applications in modern, high-level frameworks to compose applications to achieve one of several chosen measures of fairness;
- We demonstrate responsiveness of composed applications to traffic changes at timescales of a few seconds on substantial topologies, by leveraging precomputation to generate paths that support fast traffic reconfiguration when needed; and
- We implement these advances in a system, Chopin, for the ONOS controller and demonstrate its efficiency.

We begin in §2 with a discussion of background relevant for the problem we solve, followed by an overview of our approach in §3. The important internals of our design are detailed in §4, and we discuss aspects of their implementation in §5. We empirically evaluate our implementation in §6. We discuss related work in §7 and conclude in §8.

## 2 Background and Motivation

In this section we show example use cases of composing multiple resource-management applications and highlight the shortcomings of existing work. If two applications run their respective optimizations and attempt to implement the solution naively using an SDN controller, they run the risk of interfering with each other and overloading network resources. Hence, a composition step is necessary to facilitate proper resource sharing. We broadly divide the existing composition approaches into two classes, white-box and black-box, based on how they treat the optimizations. We highlight the limitations of both, by examining individual techniques and their shortcomings.

As a concrete example, consider Fig. 2. Suppose the administrator desires to install two different applications: one to balance the load of web traffic 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 $N_1$-$N_3$-$N_5$.

**Black box composition:** We classify an approach as “black box” if the optimizations are executed separately and only their inputs are modified to produce a correct result. Two common techniques are static resource allocation and ordered optimization [9].

**Static allocation** 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, 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 \( N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \), which lacks a firewall and is longer than \( N_1 \rightarrow N_3 \rightarrow N_5 \) — resulting in a failed policy enforcement and suboptimal latency.

Ordered optimization solves problems sequentially. After the first optimization is executed, the capacity of the network is adjusted by subtracting the resources consumed and the next optimization is run using the network with residual capacities. In our example, after \( App_1 \) is run, due to link load-balancing, residual capacity of the network is identical to that of \( App_2 \) 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 [3] and Athens [4] which make applications aware of the other applications and allow them to vote on each others resource-management proposals to negotiate a fairer solution. We strive to solve the composition problem without imposing this awareness requirement on application developers. Additionally, as we will show in §6, voting approaches can also produce resource-inefficient results.

White box composition: We classify approaches as “white box” if the multiple applications are used to construct another, integrated optimization. Constructing a single optimization eliminates any discrete ordering of applications or explicit negotiation between them, resulting in more degrees of freedom when making routing decisions.

Manually re-designing the application optimization(s) for composition is a powerful method, but requires a non-trivial amount of effort and expertise, making this approach difficult to scale. It is also prone to errors, making it unsuitable in a production environment where applications are expected to work out-of-the-box.

Low-level composition merges the optimization encodings as they would be given to the solver (e.g., Gurobi). However, the low-level code (example shown in Fig. 4) has little semantic information about the topology, traffic, or network resources. For example, it is unclear what quantity the variable \( x_{4_0} \) represents, nor can it be easily mapped to a variable in another application’s optimization. Furthermore, different types of constraints have different resolution policies. For example constraint R1 is a flow conservation constraint and must not be arithmetically combined with others, while R12 is a load-computation constraint and must be “added” to load-computation constraints from other applications. Naively merging low-level optimizations has no clear meaning.

High-level composition leverages network optimization frameworks such as SOL [17], Merlin [40], and Maple [44]. Composing applications written in these frameworks is viable because these frameworks retain semantic information about the optimization that could permit the automatic reconciliation of multiple applications’ specifications, making it an app-agnostic and potentially robust approach.

Committing to a composition strategy leveraging high-level frameworks leaves multiple options for how to calculate the composition, however. In order to ensure that applications can be responsive to traffic changes and other events, frameworks like SOL offload a significant portion of the computation to an offline step, where the online part of
Figure 5: Uncoordinated path selection can result in no available solution. Both applications choose shortest paths (in bold), sufficient per application, but lack capacity in a composition scenario.

<table>
<thead>
<tr>
<th>Approach</th>
<th>App-unaware</th>
<th>Fair</th>
<th>Responsive</th>
<th>Resource-efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-box</td>
<td>Static allocation: Ordering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Voting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White-box</td>
<td>Low-level</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>High-level</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chopin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Automated composition approaches and desired features. Filled circle indicates satisfactory result. Unfilled circle indicates unsatisfactory result or unknown implementation.

coordination during preprocessing (which is needed for responsiveness) can lead to resource-inefficiency or even infeasibility, as discussed above.

3 Overview

We draw a distinction between the application developer and the network operator (recall Fig. 1). Developers write their optimizations by specifying the traffic and resources they desire to manage and their optimization goals. Our overarching goal is to provide fair, resource-efficient, and responsive compositions of resource management applications while maintaining APIs that do not expose composition to the developer. The operator, in contrast, configures the composition of applications (e.g., specifying what fairness metric to use) and global network constraints (such as maximum utilization of each resource). Chopin combines demands from the developers and the operator, constructs and solves a unified optimization (using a linear programming solver), and produces a solution that can be deployed using an SDN controller.

3.1 High-level approach

In light of the discussion in §2, we begin by choosing the high-level, white-box approach. We adopt the path abstraction proposed in recent work [17] as a general, unifying “language” upon which the unified optimization is constructed. We ensure that paths are selected in a coordinated manner, specific to the set of applications being composed, by introducing an offline, coordinated path-selection step.

Unfortunately, multiple applications exacerbate the challenges associated with offline path selection. Specifically, they add another level of complexity to the optimization, since an application’s routing decisions are reflected in the network state, which in turn drives the decision making of co-located applications. Consider the network in Fig. 6, where a traffic volume shift occurs between time epochs $e_1$ and $e_2$. Despite the total volume of traffic being the same, $App_1$ cannot route traffic according to policy as path $N_1 \cdot N_5 \cdot N_6$ is not chosen, despite the total volume of traffic remaining the same.

Figure 6: Traffic shift from time epoch $e_1$ to $e_2$ causes a policy violation if path $N_1 \cdot N_5 \cdot N_6$ is not chosen, despite the total volume of traffic remaining the same.

the problem is “trimmed down” to be solvable quickly.

Failing to account for composition in the offline part of the computation can result in a final solution that is resource-inefficient. To see this, consider SOL’s offline step, which involves selecting a subset of network paths over which the online optimization will be performed. For example, both SOL and Merlin support, and suggest, computing shortest paths offline and then performing online optimization only over these paths. However, in a multi-app scenario, this can lead to infeasible or resource-inefficient solutions. Consider a network in Fig. 5: two applications are required to choose two paths, and they pick the shortest available. While sufficient for each application on its own, the total capacity of the paths is too low to carry traffic from both applications.

Summary: A summary of automated approaches and their features is given in Table 1. Black-box approaches either produce resource-inefficient results or, in the case of voting, require developers to be application-aware. (Voting also lacks responsiveness, as we show in §6.) We are aware of no low-level white-box composition techniques, nor do we know how to implement composition at a low level, and so we have indicated this hypothetical alternative as unsatisfactory across the board. Existing high-level frameworks provide the semantic information needed to compose applications’ objectives fairly (as we do here), but their lack of
was not selected during preprocessing. Multi-app path selection needs to account for such potential traffic shifts between applications to avoid infeasibility pitfalls.

To remedy this, we update the coordinated selection to choose paths that are tolerant to traffic variations. If the traffic pattern shifts at a later time, only small flow allocation adjustments will be necessary, without the need to re-select the paths. Performing coordinated path selection is computationally difficult due to the large number of paths in the optimization and the high dimensionality of the traffic matrix. We introduce two heuristic scalability improvements (described in §4.3): traffic matrix clustering and simulated annealing, aimed at maintaining the tractability of the offline coordinated path-selection problem.

3.2 Workflow

**Application development:** The developers express their optimizations using a declarative application model. They specify the type of traffic their application manages, an objective function, and how the traffic consumes network resources. For example, an application for minimizing latency of SSH traffic (recall §2) would specify the following:

- **Traffic** All SSH traffic
- **Objective** Minimize latency
- **Constraints** Route all traffic
- **Resource costs** Bandwidth: 1KB per flow
- **Policy** Path contains firewall

Since Chopin adopts a path-based optimization model, the declarative model can support a variety of network management applications expressible using paths (e.g., [26, 15, 33, 16, 38]). Resource consumption per flow (e.g., for modeling bandwidth usage), per path (e.g., for modeling TCAM constraints) and other types of path-based constraints can be expressed in Chopin, though this is not the focus of our work. We refer the reader to work by Heo and others [17].

**Offline coordinated path selection:** The operator collects the applications she wishes to deploy and generates network paths that conform to the applications’ policies (step ➊ in Fig. 7). We assume that policy conflicts between applications can be resolved prior to running Chopin (e.g., with a tool like PGA [32]). She specifies global resource utilization limits, and the type of fairness with which to combine the applications. She then generates a collection of traffic matrices, one per epoch (step ➋), which is used as an input to the path-selection process. The temporal variability of the traffic matrix across epochs dictates the robustness of the solution and can be generated from past observations or using synthetic models (e.g., [43]). The coordinated path selection (step ➌) 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 in the path store for use in the online deployment phase.

4 Detailed Design

In this section we detail the workings of selected steps in the workflow of Fig. 7. We focus on the steps that represent our primary technical innovations, namely step ➌ (described in §4.2) and then step ➍ (described in §4.3). We begin in §4.1 with defining terminology that facilitates these discussions.

4.1 Preliminaries

Composition is performed by combining individual elements of the applications in a systematic way. Each application declares its traffic classes and policy, which are used to generate valid network paths. 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 $c$.in, exiting at a designated egress node $c$.out, and matching the specification $c$.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 $c$.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-grained classes, e.g., [32].) A traffic matrix $TM_e$ for epoch $e$ holds the value

$$\sum_{c \in \mathcal{C} : \text{c.in} = \text{in} \land \text{c.out} = \text{out}} \text{c.vol}[e]$$

at location $TM_e[\text{in}, \text{out}]$.

**Applications:** For our purposes, an application $\text{App}$ is specified as a set of traffic classes $\text{App.classes} \subseteq \mathcal{C}$ that it manages; a set of permissible paths $\text{App.paths}[\text{in}, \text{out}]$ for carrying traffic classes $c \in \text{App.classes}$ such that $\text{c.in} = \text{in}$ and $\text{c.out} = \text{out}$; an average per-flow amount $\text{App.cost}[r]$ of resource $r$ consumed by traffic associated with this application; an objective function $\text{App.obj}$ 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 $\text{App.classes}$ to the network. The set $\text{App.paths}[\text{in}, \text{out}]$ is generated in step 1 of Fig. 7 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 $p \in \text{App.paths}[\text{in}, \text{out}]$ has a fixed resource-$r$ capacity $N.\text{cap}[r]$, specified in the same units as $\text{App.cost}[r]$. Similarly, each link $L$ on path $p \in \text{App.paths}[\text{in}, \text{out}]$ has a fixed resource-$r$ capacity $L.\text{cap}[r]$. It is convenient to define the per-flow cost for resource $r$ associated with traffic class $c$ to be

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

and the allowable paths for $c$ to be

$$\text{c.paths} = \bigcap_{\text{App}: \text{c} \in \text{App.classes}} \text{App.paths}(\text{c.in}, \text{c.out})$$

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. For example, decision making for middlebox load balancing and link load balancing occurs simultaneously. Fig. 8 provides a conceptual view for the composition process: a single online optimization is constructed from the application building blocks provided using the declarative model. A fairness measure is applied to the applications’ objective functions. Application-specific constraints (such as flow conservation constraints) are combined unmodified, while resource load constraints are computed from the traffic classes and resource costs provided by each application. Fig. 9 describes the mathematical underpinnings of the optimization. We emphasize that for this subsection, $e$ is a constant, and $E$ denotes the singleton set $E = \{1, \ldots, \text{NumEpochs}\}$ in the offline path selection. Offline path selection also adds Eqn. 7–Eqn. 9.

**Resource load and objectives:** To standardize resource consumption, all resource-$r$ loads and objectives must be 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. (We describe how to compute available paths in §4.3.) For example, for resources relevant to nodes, we define the resource-$r$ load $N.\text{Load}[r, c]$ induced by traffic class $c$ on node $N$ during

$$\maximize \quad \text{Obj} = \sum_{i} w_i \times \text{App_i.obj}[E]$$

subject to, for all $e \in E$,

$$\sum_{p \in \text{paths}} x_{c,p,e} \frac{c.\text{cost}[r] \times c.\text{vol}[e]}{N.\text{cap}[r]}$$

$$0 \leq x_{c,p,e} \leq 1$$

$$N.\text{Load}[r, c] = \sum_{c \in \mathcal{C}} N.\text{Load}[r, c]$$

$$N.\text{Load}[r, c] = \max_{N} N.\text{Load}[r, c]$$

$$0 \leq x_{c,p,e} \leq b_{c,p}$$

$$b_{c,p} \in \{0, 1\}$$
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.

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 \( 1 - App.obj \) as the new \( App.obj \). For example, a maximization objective that minimizes the load on node resource \( r \) is

\[
App.obj[N][E] = 1 - \max_N \sum_{c \in App.classes} NLoad_c[N][r][e]
\]

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, Chopin is capable of supporting a variety of fairness metrics (also known as welfare functions). Two natural ways of enforcing fairness are at the objective level and at the resource level. 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.

For the 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.

Another common linear metric is maximizing the minimum objective (i.e., Rawlsian difference principle [14]):

\[
\text{maximize} \ Obj = \min_i App_i.obj[N][E]
\]

At the price of higher computational cost (due to their quadratic nature), the relative mean deviation and variance functions [2] can be supported. As a special case, Chopin supports proportional fairness [20]— a commonly used fairness metric. Proportional fairness is defined as:

\[
\text{maximize} \ Obj = \sum_i \log(App_i.obj[N][E])
\]

Since a log function cannot be directly incorporated into a linear program, Chopin implements a piece-wise linear approximation, based on work by Camponogara et al. [7].

### 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 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. 10).

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

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

(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 per-
form offline path selection. Similarly, an increase in the network size and so the 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 simulated annealing, shown in Fig. 11 and 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 \(\{\langle c_1, \text{vol}(e), c_2, \text{vol}(e), c_3, \text{vol}(e), \ldots \rangle\}_{e \in E}\) using Ward’s hierarchical clustering [46], into a specified number \(\text{NumClusters}\) of clusters. The centroids of these clusters then are used as \(\text{NumClusters}\) “epochs” in the path-selection formulation, thereby improving its performance. We choose Ward’s clustering for two reasons: it allows specification of precise number of clusters (i.e., \(\text{NumClusters}\)) and scales well to a large number of samples (i.e., \(\text{NumEpochs}\)).

**Simulated annealing speedup:** 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. To combat this, we employ a simulated annealing (SA) approximation to find sets of paths (i.e., the \(b_{c,p}\) variables) that improve the overall objective. At a high level, SA avoids solving a large ILP by iteratively sampling from the available set of paths, running the optimization over the sampled paths only (conceptually, fixing \(b_{c,p} = 0\) for other paths \(p\)), and adopting the chosen paths as the “current best” if the optimization improves on that obtained from previously chosen paths or, with some probability, even if not.

More specifically, SA works in our context by first selecting a number \(\text{NumPaths}\) per class \(c\), and using these \(|C| \times \text{NumPaths}\) paths as the only allowable paths in the unified offline optimization (as described above). Each unused path in the solution is then replaced by a new path with the same ingress and egress node, and the composed optimization is solved again, using only these new paths (plus the paths previously used) as the allowable paths. The new solution is retained with probability 1 if it is better than the previous, or with probability \(\frac{1}{1 + e^{\frac{\text{temperature}}{t}}}\) if not, where \(t\) is a “temperature” parameter to the SA algorithm. The algorithm continues for a fixed number of iterations, adjusting the temperature in each [21], after which the retained solution is output as the final solution.

**Path replacement:** The success of simulated annealing hinges on the heuristic logic responsible for choosing paths to be used 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 choice of paths favors shorter paths and, among paths of the same length, 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), and yet provides sufficient freedom to load balance the applications’ resources of interest.

## 5 Implementation

**Chopin Library:** The Chopin library is built using 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 [13].

**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 implement a Chopin service in the ONOS controller. Fig. 12 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 ①). A newly deployed application registers with the service and provides its optimization requirements (step ②). This starts the re-computation process, which utilizes the REST API to communicate with the optimizer to request the composition of all applications registered up to this point (step ③). The Chopin service parses the solution received from the optimizer, generates appropriate intents and returns them to the application(s) (step ④). The service also allows the administrator to specify global network constraints that will act across applications. This architecture ensures that the Chopin ser-

---

**Figure 12:** 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.
vice conforms to ONOS’ event-driven nature and maintains applications’ unawareness of other applications.

6 Evaluation

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

- end-to-end validation using the ONOS controller (Fig. 13)
- resource-efficiency improvements over static allocation and voting approaches (Fig. 14, 15)
- resource-efficiency benefits over uncoordinated path selection (Fig. 17)
- impact of different fairness metrics have on the solution (Fig. 19)
- runtime improvements in scalability of path selection due to the clustering and annealing techniques (§6.2).

Setup: We chose topologies of various sizes from the TopologyZoo dataset [22]; 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 [1]. 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 [35] and introduced a temporal variation between applications’ traffic volumes using a Dirichlet distribution [19] across a 100 epochs. We choose 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., [43]) 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—and composed applications using a utilitarian fairness metric. All times below refer to computation on computers with 2.4GHz cores and 128GB of RAM, except deployment benchmarks, where we used Mininet [27] in a virtual machine to emulate the topologies.

End-to-end validation: We setup different topologies using the Mininet emulator and ONOS controller. We deployed up to four applications (specifically, four copies of the traffic-engineering application with disjoint traffic classes) on each topology. Fig. 13 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 timescale responsiveness.

Figure 13: Time to deploy multiple applications as a function of number of applications. Includes time for online optimization and computing and installing ONOS path intents.

Figure 14: Optimality comparison between Chopin, Static allocation, and Athens-like voting framework. “Voting annealing” uses paths pre-selected by the Chopin simulated annealing algorithm (for speed), “Voting all” uses all available paths.

6.1 Resource-efficiency, Fairness and Responsiveness

Next, we use trace-driven simulations to evaluate the benefits of using Chopin for resource-efficiency, fairness, and responsiveness and compare it to black-box approaches. We also evaluate the potential benefits of using Chopin’s coordinated path selection approach compared to prior work [17].

Resource-efficiency vs. black-box approaches: We compare Chopin to black-box approaches: naive static allocation and Athens [4] — arguably the closest practical work in this 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 be a “proposal” that is submitted for voting. Each application was allowed to submit up to three proposals and used either all available paths (denoted “Voting all”) or paths pre-selected by simulated annealing
Fig. 15: Time to compose and solve a single optimization (mean across 100 epochs) for different composition approaches and topologies.

Fig. 16: Objective function values of static allocation approaches and Chopin computed with variable number of paths. Larger path sets provide marginal improvement, but cannot compete with coordination benefits of Chopin.

Fig. 17: Relative improvement in the objective value over the base solution where all applications use shortest paths. “Annealing” indicates paths chosen by coordinated path selection with simulated annealing, “Optimal” the is optimal solution using all paths.

Fig. 18: Resource-efficiency improvements over the baseline strategy of using shortest paths, as network congestion increases.

(quoted text)

Fig. 14 shows the objective function (higher is better) for each topology and different composition approaches. Each box 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 other approaches by as much as 60%, and naive voting fails to converge in some cases (e.g., k4 and larger topologies).

Fig. 15 shows the mean time to construct an optimization and compute a solution for a single epoch. Chopin provides better responsiveness compared to the black-box approaches due to the coordinated path-selection step. We also compare Chopin to a static-allocation approach that uses a path-based optimization, but naive shortest-path selection (as outlined in [17]). The results in Fig. 16 show that the objective value obtained using Chopin is better by a factor of two. Increasing the number of selected paths provides marginal benefits to the static allocation approach, but is not sufficient to reach the resource-efficiency obtained using Chopin’s coordination.

Benefits of coordinated path selection: We compared solutions obtained using shortest paths, paths chosen by coordinated selection and all paths (optimal). Fig. 17 shows the relative improvement over the baseline objective computed using shortest paths. Paths chosen by simulated annealing result in a solution that is as good or better than using shortest paths, while also obtaining solutions where optimal failed to complete due to resource limits (e.g., Arn and k6 topologies). For clarity, not shown in Fig. 17 is comparison to a random path selection strategy proposed in prior work [17], which performs quite poorly in the multi-application scenario and in some cases fails to produce a result.

To further explore benefits of coordinated path selection, we generated a series of 100-epoch traffic matrices for the Abilene topology with each traffic matrix having an in-
increased total volume of traffic, to model different stages of network congestion. Fig. 18 shows the relative improvement over using shortest paths, as a function of network load. Each point is the mean value across 100 epochs (with error bars indicating standard deviation). Above 55% congestion, both optimal and Chopin start outperforming shortest paths; there is no gap between Chopin and optimal solution until congestion nears 90%, where optimal manages to find better solutions.

Impact of fairness metrics on objectives: To explore the effect of different fairness metrics on the objective functions of individual applications we composed two applications using three fairness metrics: utilitarian, max-min, and proportional (see §4.2). Fig. 19 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. Both utilitarian and proportional fairness maximize the global objective, but do so at a cost of application inequality (e.g., favoring the link load balancing on the k4 topology). This result highlights that Chopin is flexible and gives the operator ability to customize the solution according to their needs, be it overall network resource-efficiency or fairness.

6.2 Scalability

Path selection benchmarks: To show runtime benefits of using simulated annealing to select paths, we performed path selection using the optimal ILP described in §4 and simulated annealing. Simulated annealing allows orders of magnitude faster offline path selection (Fig. 20) while returning the same quality of paths as the ILP selection (Fig. 21), for topologies where ILP selection could complete.

Clustering benchmarks: Similarly, we demonstrate the impact of traffic matrix clustering on path selection performance. Fig. 23 shows the time to select paths with and without traffic matrix clustering. Enabling clustering results in an order of magnitude faster runtime, due to reducing the number of epochs from 100 epochs to 10 epochs, and thus the size of the offline path-selection optimization.

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. 22 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

Figure 19: Impact of chosen fairness metric on the objective function of each application

Figure 20: Runtime comparison of the optimal ILP path selection and simulated annealing selection

Figure 21: Resource-efficiency comparison of the optimal ILP path selection and simulated annealing selection
showed that naively extending these frameworks to multi-application use cases presents new challenges, namely lack of fairness and resource-efficiency. Our work addresses these limitations, proposes a new system design that separates application development and deployment concerns, and produces fair, resource-efficient solutions.

Robust optimization: The field of robust optimization [6] develops ways of “protecting” the optimizations against uncertain data. In networking this takes on the form of network design validation against failures [8] or (semi-)oblivious routing [5, 23, 25]. Such techniques can be overly conservative and computationally intensive, especially with multiple resources involved in addition to bandwidth. However, they can be incorporated into the internals of Chopin, possibly freeing the operator from the burden of generating sufficiently varied traffic matrices. We leave exploration of such techniques to future work.

Multi-objective optimization: Extensive literature exists on multi-objective optimization, covering a number of techniques [41, 9]. Their focus, however, is not on network optimization and often requires manual composition. Our work is specific to automatic composition of networking applications.

8 Conclusions

A growing number of SDN resource-management applications necessitates a new way of composing them that guarantees fair and resource-efficient solutions, while maintaining responsiveness to network changes. Our goal is to provide a system capable of achieving such composition while abstracting away many of the low-level composition details from application developers and network operators. In this paper, we presented Chopin — a composition framework for SDN applications that leverages high-level network optimization tools to achieve application-agnostic, fair, resource-efficient, and responsive composition. We showed that Chopin achieves significantly better robustness than previous “black box” approaches, can be integrated with modern SDN controllers, and outperforms uncoordinated approaches to composition.
9 References


