Dandelions: A Novel, High-Level Programming System for Software Defined Coalitions with Local State Sharing

Xin Wang[†], Qiao Xiang[‡], Y. Richard Yang[‡],
[†]Tongji University, [‡]Yale University,
13xinwang@tongji.edu.cn, {qiao.xiang, yry}@cs.yale.edu,

***Abstract*—Integrating software defined networking into military coalition to realize a software defined coalition (SDC) infrastructure is non-trivial due to the stringent requirements (*e.g.*, real-time, efficiency and reliability) of SDC applications in highly dynamic tactical network environments. Recent designed SDC datapaths improve the performance of SDC applications by offloading complex, stateful operations from the SDC control plane to data plane devices. However, they have two limitations. First, results of offloaded operations (referred as *local state*) are not shared between data plane devices, resulting in substantial performance issues in SDC networks. Second, configuring low-level SDC datapaths is time-consuming and error-prone. To address these two issues, in this paper, we design Dandelions, a novel, high-level SDC programming system. Dandelions provides novel high-level primitives for users to specify the behavior of SDC data plane devices. It adopts a novel decision graph data structure and an efficient configuration framework to translate high-level SDC programs into efficient SDC datapath configurations. The translated configurations allow data plane devices to exchange local states with the goal of efficiently utilizing the resources in SDC networks (e.g., the packet processing capability of devices and the data transmitting capability between devices). We implement a prototype of Dandelions and demonstrate its efficiency and efficacy using experiments. Results show that with Dandelions, the total throughput can increase around two times compared with the value without Dandelions.**

I. Introduction

The recent success of software defined networking (SDN) systems [1], [2], [3] motivates the efforts to integrating SDN into military coalitions to realize an efficient, agile, and optimal software-defined coalition (SDC) infrastructure [4]. In SDC, autonomous coalition members operate under highly dynamic tactical network environments with resource constraints, such as limited power and processing capability, and dynamic connectivity.

One may think the integration of SDN into SDC is straightforward, because SDN allows coalition members to realize efficient, flexible control over the coalition networks through flexible packet match-action processing on the data plane and logically centralized in the control plane [5], [6]. Despite these promising features, they are insufficient for supporting SDC.

The fundamental reason behind this insufficiency is that SDC applications (*e.g.*, flexible load-balancing and detection of DNS amplification attack) have stringent performance requirements, *e.g.*, real-time, efficiency and reliability, under *highly dynamic tactical network environments*. Implementing SDC applications using the above SDN architecture in such environment would incur high communication overhead between the data and control planes of SDC applications, resulting in significant delay, efficiency and reliability issues.

Some SDC datapaths are recently proposed, which offload *complex, stateful* operations on packets (*e.g.*, counting, flow security inspection and congestion control) from the control plane (*e.g.*, a base station) to data plane devices (*e.g.*, mobile devices) to improve the performance of SDC applications [7], [8]. Different datapath primitives, such as state counter, packet buffer and packet in-network processing block, are designed in these systems to support various SDC applications, *e.g.*, stateful firewall, proactive routing protection, and resilient routing.

However, these systems suffer from two limitations. First, results of offloaded stateful operations at data plane devices, which we refer to as *local state* in the remaining of the paper, are not shared between data plane devices, resulting in substantial resource under-utilization in SDC networks. For example, a firewall middlebox is usually a bottleneck in an SDC network due to its limited processing speed. As such, once a data flow is identified as secure, non-sensitive by a firewall, all future packets of this flow should be forwarded along a path with higher reliability and throughput, without the need of passing the firewall again. However, the security state of this flow is only stored in the firewall and not shared with other devices. Without knowing this flow is non-malicious, an upstream device (*e.g.*, a gateway router) in the SDC network still has to forward all the packets of this flow to the firewall. Second, although some distributed update primitives provides interface for sharing of local state between data plane devices [9], configuring such low-level SDC datapaths still requires time-consuming and error-prone manual efforts.

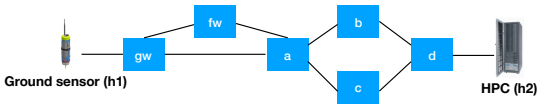
Toward address these two limitations, in this paper, we design Dandelions, a novel, high-level SDC programming system. Specifically, Dandelions introduces a series of novel programming primitives for users to model and specify the behavior of SDC data plane devices. In addition, we design a novel data structure called *decision graph* and an efficient configuration framework in Dandelions to translate high-level SDC programs into efficient SDC datapath configurations. The translated configurations allow data plane devices in SDC to exchange local states with the goal of efficiently utilizing the resources in SDC networks (*e.g.*, the packet processing capability of devices and the data transmitting capability between devices) to support real-time, efficient, and reliable performances requirements of SDC applications. We implement a prototype of Dandelions and demonstrate its efficiency and efficacy using experiments. Results show that with Dandelions, the total throughput can increase around two times compared with the value without Dandelions.

The rest of this paper is organized as follows. Section II

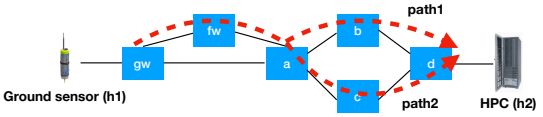
discuss related work of SDN/SDC programming and give an motivating example. Section III gives an overview of DSDC, including its programming model, architecture and workflow. Section IV presents the details of how Dandelions translate a high-level SDC program into efficient, dynamic low-level SDC data plane configurations. We evaluate the performance of Dandelions in Section V before concluding the paper in Section VI.

II. Motivation and Related Work

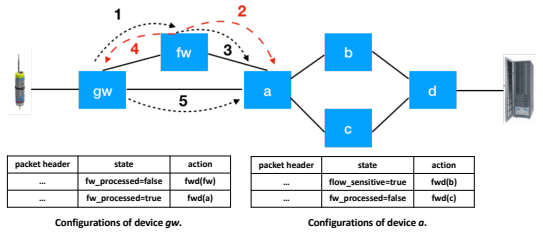
Related work: There are some related work (*e.g.*, [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]) for the offloading problem. SNAP [10] targets to deploy the state operation into datapaths from a high-level program. And it does not consider sharing local state among switches. Both [11] and [12] only focus on the path computation with constraints and do not consider packet forwarding relying on the results of middleboxes. MP-HULA [13] and Hula [14] use the probe packets to update the state variables for load-balancing which are motivating examples of Dandelions but they do not consider these complex dependencies of middleboxes. P4CEP [19] focuses on offloading event processing to a P4 switch without concerning packet transfer.



(a) Firewall example topology.



(b) Configuration of state-of-the-art.



(c) Configuration of Dandelions.

Fig. 1: A running example with a firewall as middlebox.

Motivation: We will motivate the Dandelions with a firewall example (as a running example in this paper) as shown in Fig. 1a. The topology of the firewall example consists of a ground sensor (h_1), a middebox (*i.e.*, firewall, fw), a high-performance computing server (h_2), a gateway gw, and several switches (a-d). The data collected from the sensor should first send to the firewall fw which can identify whether the data is sensitive (*SENSE*) or not (*NORMAL*) based on the 5-tuple of the packet carrying the data (*i.e.*, srcIP, dstIP, srcPort, dstPort, protocol). And the network operator wants to enforce the following policy: For any data from the sensor h_1 to the HPC server h_2, if the data is sensitive, then it should pass through the switch b, otherwise pass through c. Such policy can be specified by a high-level program as the following (**the firewall program**):

```
L1: mFW <- middlebox("firewall")
L2: path1 <- H -> b -> H //H: host, c: waypoint
L3: path2 <- H -> c -> H //H: host, d: waypoint
L4:
L5: def onPacket(pkt):
L6:     if pkt.srcAddr = h1 & pkt.dsrAddr = h2:
L7:         if mFW.handle(pkt) = SENSE:
L8:             return path1
L9:         else:
L10:            return path2
L11:    else: return DROP
```

The firewall program follows the SNAP syntax and has the following extensions: 1. Add packet handling functions for available middleboxes in the network. For example, line 1 defines a middlebox variable mFW with "firewall" as the name to indicate the middlebox fw in the network; line 7 invokes the packet handling function of mFW which can return the result of the identification of the flow (*i.e.*, the 5-tuple of the packet); 2. Support route algebra as the return. The route algebra [21] is used to give the constraints of a path in the network. For example, line 2 (3) specifies the path must pass through b (c) as its waypoints constraint.

Given the firewall program and the topology, the problem is how to setup the configuration of datapaths (*e.g.*, the flow rules in switches) to enforce packets are handled as the program specifies. As state-of-the-art of compiling SDN programs to datapath configuration, SNAP computes paths for packet forwarding based on the OBS (One-Big-Switch) model. (The location of sources and destinations of generated paths relies on hints from network operators who specifies the mapping from ip addresses to ports in the OBS [10].)

The configuration result of SNAP is shown in Fig. 1b. Specifically, $path_1$ ($path_2$) is used to forward packets that are identified *SENSE* (*NORMAL*) by the firewall. The result gives correct paths for packets from h_1 to h_2 based on the firewall program.

Though SNAP can give a correct configuration, it does not fully utilize the network resource. Specifically, considering the stateless property of the firewall middlebox (*i.e.*, the identification is only based on the 5-tuple of packets), after the identification of a flow, the result can be "cached" in switches and the following packets with the same 5-tuple can get the result.

Fig. 1c gives the design which leverages this local state sharing. We assume the stateful switches logically have two tables: one is a state table (*i.e.*, $match \rightarrow state$) and the other is a match table (*i.e.*, $match + state \rightarrow action$). A packet first enters the state table to get its corresponding state (by matching its packet fields) and then enters the match table to get the action. By allowing the local state sharing, the firewall can share its result for a flow by installingl/modifying rules in the state table. For example, it can install a rule: $dstPort = 22 \rightarrow PROCESSED$ to the state table which indicates a packet for tcp port 22 does not need to forward to fw.

The design has 5 steps: 1. The first packet arrives at the gateway gw and then is forwarded to fw (by getting a state *NOT-PROCESSED* at gw); 2. fw identifies the packet as *SENSE* and then modifies its state at a to *SENSE*; 3. fw sends the packet to a where the packet gets state *SENSE*; 4. fw modifies the packet's state at gw to *PROCESSED*; 5. The following packets with the same match fields as the first packet enter gw and then are all forwarded to a. Compared with the

result of SNAP, this design can obtain a higher throughput as flows do not need to pass through the firewall except their first packets. However, the design not only needs to give the paths but also the behavior of the firewall, *i.e.*, modifying what state at which switches (like step 2 and 4). Then, the problem is: Given a program and a topology, how to automatically setup the configurations of switches and middleboxes (*i.e.*, their behavior) to improve the performance if local state sharing ~~is allowed~~?

III. Dandelions Overview

In this section, we will first give the programming model of Dandelions and then show the architecture and workflow of ~~Dandelions~~ that transforms a program to datapath configuration including the behavior of middleboxes.

A. Programming Model

As described in Section II, we adopt the basic syntax of SNAP that consists of predicates (*i.e.*, test/read packet's match fields or state variables) and policies (write values to packet's match fields or state variables) but with the following middlebox related primitives: 1. Specify a middlebox in the network (`m = middlebox(name, property)`) where the property indicates the middlebox is stateless or not; 2. Invoke the packet handling of a middlebox (`m.handle(pkt)`). Specifically, we consider a middlebox as a packet handling function that can return the result for the incoming packet. For the stateless middlebox, if two packets have the same 5-tuple match fields, then they always have the same results. For the stateful middlebox, this cannot be guaranteed as its packet processing depends on the state in the middlebox.

And to handle the returned result of a middlebox for a packet, we consider the following switch-middlebox interaction process. A switch can send a packet to a middlebox for the processing, and after the processing the middlebox can send the packet back and share its local state for the flow to switches (leveraging their stateful datapath design). Note that by using tunnels, any switch can send a packet to any middlebox, and also the inverse. Though the interaction is simple, there are several details need to consider (*e.g.*, in the step 3 of the Dandelions design for the firewall example, how to guarantee that the packet gets the newest state?). We will talk about the details in the next section.

Besides the middlebox primitives, we also support the route algebra as the return to specify path constraints for packets in our program (*e.g.*, line 2 and 3 in the firewall program). Therefore, we consider the three kinds of statements (besides the basic control statements such as `if, else`): 1. Test of packet match fields; 2. Packet handling of a middlebox; 3. Route algebra as a return.

B. Architecture and Workflow

The architecture and workflow is shown in Fig. 2. Step 1: Compute the Decision Graph (DG) from the program (and the definition of DG will be shown in the next section). Step 2: Based on the DG and the topology, compute optimal paths (*i.e.*, concrete paths instead of constraints) with a system defined objective (*e.g.*, maximizing total throughput). Step 3: Generate datapath configuration (including switch-middlebox tunnels) in the network.

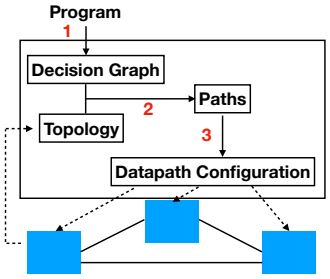


Fig. 2: The architecture and workflow of Dandelions.

IV. Design Details

In this section, we will first provide the details of DG, and then discuss the path computation based on the DG and topology. Finally, we will discuss datapath generation from the paths.

A. Decision Graph

The Decision Graph (DG) is used to capture the trace of every packet applied by the program. The trace is the decision dependencies (*i.e.*, the results of packet fields test statements and middlebox processing statements) of a packet and its returned path (*i.e.*, path constraints). In this paper, for simplicity, we only consider the waypoints constraint (*i.e.*, a sequence of nodes in the network that packets should pass through). We define a DG as a directed acyclic graph (with a single root) that has three types of nodes: packet-test nodes, middlebox-operation nodes, and action nodes. The first two nodes are the internal nodes of DG while the last one is the leaf nodes. An out-edge of a node can specify a range of packets (if the node is a packet-test node) or a result from a middlebox (if the node is a middlebox-operation node). For example, the DG of the firewall program is shown in Fig. 3. Note that we do not focus on the computation of DG from a program which can be achieved by existing (compiler) work [10] [22].

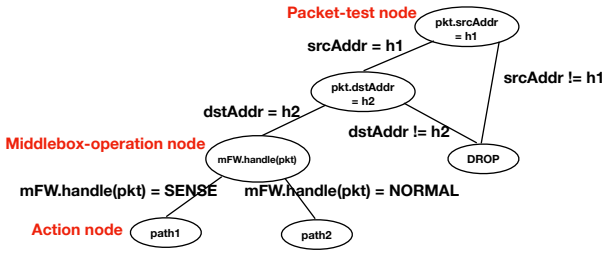


Fig. 3: Example of DG.

Given a DG, we denote the sequence of nodes and edges from the root to an action node (*i.e.*, path constraint pc) as the trace of pc ($T(pc)$). Then, given a trace $T(pc)$, we denote the middlebox-operation nodes of $T(pc)$ as $M(pc)$ (*i.e.*, a sequence of nodes). As middlebox-operation nodes represent the packet handling of middleboxes, to not form loops, every middlebox-operation node can only appear once in $M(pc)$. Also, by extracting middlebox-operation nodes that represent stateful middleboxes from $M(pc)$, we have a subsequence of $M(pc)$ that every node is for stateful middlebox operations (denoted by $M^s(pc)$).

B. Path Computation

As the leaf nodes in DG are path constraints, path computation targets to compute the concrete paths for these leaf nodes

system performance

with a performance objective.

Packet forwarding model: Before path computation, we first give a packet forwarding model with DG in the network as the following. As shown in Fig. 4, given a path ($gw \rightarrow a \rightarrow b$) in the firewall example, logically we consider every switch in the path has the DG of the firewall program where P indicates a packet-test node, M indicates a middlebox-operation node, and A indicates an action node. The A node with red color represents the corresponding constraints for the path. The first packet of the flow (arriving at gw) starts to traverse the DG as the red line in the figure. When the packet meets a middlebox-operation node, it will be sent to the corresponding middlebox through a tunnel. After the processing, the middlebox will send back the packet and share its local state (*i.e.*, set the state) for the flow with the corresponding middlebox-operation node at every switch along the path. (If a middlebox is stateful, then it does not set any state but make the packet carry the state.) After the traversal of the DG (*i.e.*, arriving at the leaf node along the red line), the packet will get the path and will be forwarded along the path. As local states have been shared with corresponding middlebox-operation nodes (or carried by the packet for stateful middleboxes) at other switches, the packet does not need to be sent to the middlebox again at the next switch. ~~As the local states of all the stateless middleboxes for a packet can be sharing with stateful switches,~~ we say a forwarding of a packet is a stable forwarding if the packet does not access any stateless middlebox (compared with the first packet that should access them).

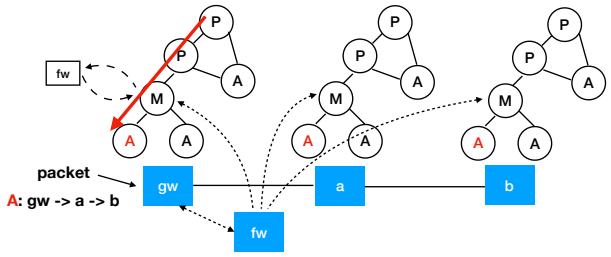


Fig. 4: Packet forwarding model with DG.

Initialization: As the initial process, for every action node in the DG, system should identify its source and destination nodes in the network. (This can be achieved by the hints from network operators as [10].) Then, add the source and destination nodes to the corresponding waypoints constraint, and also the $M^s(pc)$ as the processing of stateful middleboxes should have a correct order. We consider the waypoints constraint as a set of node pairs (x, y) which indicates packets must pass through x before y. (Note that x and y can be source/destination nodes.)

System objective: Based on the model, as the forwarding of the first few packets has little influence on the performance of its flow, a simple path computation (that only considers the stable forwarding) would be the following. We consider the objective of the network is to maximize the total throughput, *i.e.*, maximize $\sum_i b_i$ where i is the index of the path constraint pc_i and b_i is its throughput. The definition of variables can be found at Table I. And the constraints can be found at Table II (where $Path(x, y, E)$ means there exists a simple path starting from x to y in E).

However, this simple solution can get optimal paths only if all the middleboxes are stateless. If there is a stateful middlebox, the constraints of stable forwarding should not only consider the waypoints constraints specified by the programmer as it also must pass through the stateful middleboxes.

Variable	Description
u_i, v_i	The source and destination nodes of path constraint pc_i
E	All edges (an edge e: $(e.src, e.dst)$) in the network
m_e	The maximum bandwidth of e
W_i	A set of node pairs of pc_i
z_e^i	The edge e is selected by pc_i
b_i	The bandwidth of flows using the path for pc_i

TABLE I: Notation.

Constraint	Description
$\forall i, \forall (x, y) \in W_i, Path(x, y, E)$	Path constraints
$\forall i, Path(u_i, v_i, E)$	Path exists in E
$\forall e \in E, \sum_i b_i * z_e^i \leq m_e$	Bandwidth constraints for edges

TABLE II: Constraints.

Then, people may add another constraint as the following: $\forall (x, y) \in M^s(pc_i), Path(x, y, E)$. This constraint enforces that the optimal paths must pass through all the stateful middleboxes in a correct order (*i.e.*, $M^s(pc_i)$). This seems correct as now the stable forwarding includes the stateful middleboxes.

not enough

However, this is wrong. Considering the example as shown in Fig. 5. As the middlebox M is stateful, the new added constraint enforces that the computed optimal paths should consider M. However, the result could be that M is behind the branch of waypoints b and c. In this case, when a packet arrives at a, it cannot decide how to forward to M as which path to select depends on the result of M.

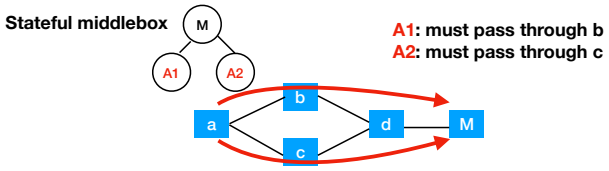


Fig. 5: An example to show that the dependencies between middlebox-operation nodes and waypoints should be considered.

To resolve the issue in Fig. 5, one solution is to add constraints (M, b) and (M, c) to A_1 and A_2 respectively. Then the paths in Fig. 5 cannot exist as they does not satisfy the constraints. The idea is to enforce a packet must get results of stateful middleboxes and then select the correct path.

Though the idea is simple, a straightforward implementation may cause excessive constraints for paths. For example, still a stateful middlebox-operation node (M) has two path constraints A_1 and A_2 as its children in a DG. And, A_1 specifies a path must pass through a, b; A_2 specifies a path must pass through a, c. Based on the idea, we can add (M, a) to both A_1 and A_2 waypoints constraints, and the computed paths are guaranteed to meet the constraints. However, this is only a sufficient condition as some other conforming paths are excluded (*e.g.*, $a \rightarrow M \rightarrow b/c$).

Waypoints constraints computation: Now we give an algorithm to compute the waypoints constraints that enforce all the paths are correct and no conforming paths can be excluded. The high-level structure of the algorithm is to do a depth-first traversal for a DG. The traversal only considers the stateful

middlebox-operation nodes and action nodes (and traverses a stateful middle operation node only after all its children are finished). When traversing a stateful middlebox-operation node, we can get all possible waypoints constraints under its decision. Then, the processing of the node is to update these constraints (each of which can be modeled as a directed acyclic graph, DAG). As discussed before, we do not want to add excessive constraints. And the solution is simple: When processing a middlebox node M, if all the no-incoming-edge nodes of M's possible constraints (*i.e.*, DAGs) are the same, then skip these nodes recursively until these nodes are not the same and then add edges (M, x) to each of DAGs where x is the node of the location where the skipping process stopped. The insight is that, skipping a node is safe if and only if the node is the next waypoint node for *all* the possible paths. For the example in Fig. 6, the node M has four possible path constraints (DAGs) and a should be skipped when updating them.

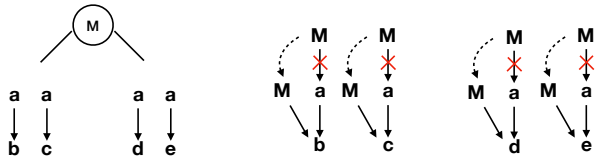


Fig. 6: Skip the same nodes.

C. Datapath Generation

Remove redundant nodes of DG: After the path computation, there are concrete paths for all the leaf nodes in DG. We follow the packet forwarding model described previously that all the switches in the network have the same DG. It is easy to observe that if a switch does not belong to a path, then the corresponding leaf node of the path can be removed (and then the no-out-edge internal nodes). Also, by converting the path in a leaf node to the next hop in the network, we may still remove redundant nodes. Still consider the firewall example. After we convert the paths to next hops, we find that the next hops of two leaf nodes are the same, *i.e.*, switch a in the network. This means the middlebox-operation node has no meaning for the gateway gw since whatever the result of the node is, the next hop does not change. Therefore, we can remove the node and replace with the leaf node with the next hop a. Fig. 7 illustrates the process.

Then, for the switch side, based on the trimmed DG, it can generate tables and flow rules easily with the multi-table pipeline structure. Note that a middlebox-operation node can be viewed as a state table following a match table. As for the middlebox side, if it is stateless middlebox, then the only datapath it needs to care about is switch-middlebox tunnels which also can be easily generated. For the stateful middleboxes, as they belong to computed paths, they can be viewed as switch nodes with stateful functions that can embed the results to the packets. When a packet meets a stateful middlebox node in DG of a switch, and the node cannot be replaced with a leaf node, then any next hop under the node is acceptable since it must eventually arrive at the middlebox and any path from the switch to the middlebox must follow the path constraints.

Order of messages: The next issue is how a stateless middlebox share local states (*i.e.*, update states) in other switches.

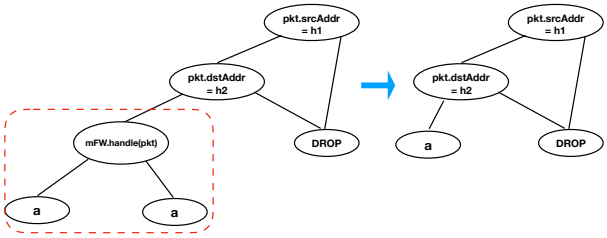


Fig. 7: Remove redundant middlebox nodes.

It needs to guarantee that for any targeting switch, the update message should arrive earlier than the packet. For example, in the firewall example, only the step 2 is finished, step 3 can be executed. Then a simple solution can be making the packet carry the update message. Different with the state carrying for stateful middleboxes, this message can install/modify rules in the state table of the corresponding middlebox-operation node.

V. Performance Evaluation

In this section, we will first demonstrate the benefits of Dandelions from two aspects: latency and total throughput, and then show the benefits of the waypoints constraints computation part. All evaluations are run on an 3.5 GHz Intel i7 processor with 16 GB of RAM running Mac OSX 10.13.

Methodology: First we generate a random topology with 25 nodes and 50 edges. For every edge, we set two random values as its latency (5 - 10ms) and bandwidth (5 - 10Mbps). To model a flow in the topology, we randomly choose two nodes from the topology as the source and destination nodes of the flow. And a flow can have a sequence of nodes (other than its source and destination nodes) in the topology as its required ordered middleboxes for packet processing. (We add a constraint to the selection for these middlebox nodes that the number of neighbors of a middlebox node must equal to two as typically a middlebox does not have route selection capability, *i.e.*, for any packet, it only has one output interface.) As a comparison of Dandelions, the traditional approach does not distinguish whether a middlebox is stateful or stateless. Therefore, when computing a path for a flow in the tradition way (*i.e.*, do not apply Dandelions), it requires the path must pass through all the middleboxes in a correct order. And when computing a path for a flow in the Dandelions approach, we random choose a subset of its required middlebox nodes as stateful middleboxes since for Dandelions approach, stateful and stateless middleboxes are handled in different ways.

Latency: To show the benefits for the latency aspect, we consider a single flow and differentiate its number of required middleboxes. And the target is to find an optimal path to minimize the latency for the flow. Then, we compare the results (*i.e.*, the minimal latency) between applying Dandelions and not.

Total throughput: To show the benefits for the total throughput aspect, we consider multiple flows and all flows have the same required middleboxes. We also differentiate the number of middleboxes. And the target is to find optimal paths that have maximum total throughput. Then, we compare the results (*i.e.*, the maximum total throughput) between applying Dandelions and not.

Execution time: To evaluate the performance of Dandelions, we compare the execution time of the path computation to

	(F=5, M=1)	(F=5, M=3)	(F=5, M=3 (S=1))
With Dandelions	1.3 (s)	1.4 (s)	4.2 (s)
Without Dandelions	4.5 (s)	10.8 (s)	11.5 (s)

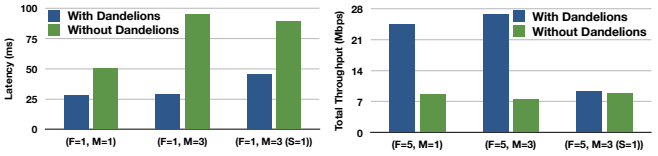
TABLE III: The execution time of path computation to maximize total throughput for different scenarios.

maximize total throughput for both approaches (*i.e.*, applying Dandelions and not).

Waypoints constraints computation: To show the benefits of the waypoints constraints computation, we set a sequence of nodes in the topology as a flow's waypoints constraint. Then, we consider the minimal latency as system's objective and compare the results between applying the waypoints constraints computation and not (*i.e.*, leading to excessive constraints). In the excessive constraints, all the middlebox nodes should be passed through before flow's waypoints.

Results: The results in Fig. 8a demonstrate that by using Dandelions, the latency can be reduced. Specifically, F specifies the number of flows; M specifies the number of required middleboxes for flows; S specifies the number of stateful middleboxes for flows. As minimizing latency for a flow does not affect the results of other flows, the experiment only considers one flow scenario. From the results, we can see that without Dandelions, the latency of the flow grows up when the number of stateless middleboxes increases but if Dandelions is applied, the latency grows up only when the number of stateful middleboxes increases.

The results in Fig. 8b demonstrate that by using Dandelions, the total throughput can be increased. Specifically, when there are 5 flows and 3 middleboxes, the total throughput with Dandelions is around 3 times of that without Dandelions.



(a) The latency for different scenarios.

(b) The total throughput for different scenarios.

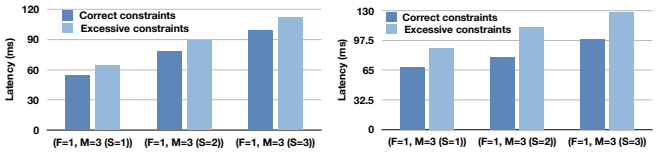
Fig. 8: The benefits of Dandelions for latency and total throughput.

The results in Table III show the execution time of the path computation part to maximize total throughput. Since with Dandelions, the number of constraints is smaller than that without Dandelions, the execution time is also reduced (from 10.8 to 1.4 seconds when F=5 and M=3).

Fig. 9 shows the latency with correct constraints (*i.e.*, by applying waypoints constraints computation) and with excessive constraints. Specifically, the results in Fig. 9a consider that the waypoints only have one node while Fig. 9b are for three nodes. From the results, we can see the excessive constraints increase the latency, *i.e.*, lead to non-optimal path computation result. And as the waypoints constraint has more nodes, the latency result with excessive constraints becomes larger.

VI. Conclusion

In this paper, we design Dandelions, a novel, high-level SDC programming system. Dandelions provides novel high-level primitives for users to specify the behavior of SDC data plane devices. It adopts a novel decision graph data structure and an efficient configuration framework to translate high-level



(a) One nodes in waypoints constraint.

(b) Three nodes in waypoints constraint.

Fig. 9: The latency with different waypoints constraints.

SDC programs into efficient SDC datapath configurations, which allow data plane devices to exchange local states with the goal of efficiently utilizing the resources in SDC networks (e.g., the packet processing capability of devices and the data transmitting capability between devices). Experiment results on a prototype of Dandelions demonstrate its efficiency and efficacy.

References

[1] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *SIGCOMM'13*.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM'13*.

[3] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: steering oceans of content to the world," in *SIGCOMM'17*.

[4] V. Mishra, D. Verma, C. Williams, and K. Marcus, "Comparing software defined architectures for coalition operations," in *ICMCIS'17*.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.'14*.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM'14*.

[7] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM'14*.

[8] Q. Xiang, F. Le, Y.-S. Lim, V. K. Mishra, C. Williams, Y. R. Yang, and H. Zhang, "Opensdc: A novel, generic datapath for software defined coalitions," in *IEEE MILCOM'18*.

[9] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "Ddp: Distributed network updates in sdn," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1468–1473.

[10] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 29–43.

[11] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using sol," in *NSDI'16*.

[12] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *ACM CoNEXT'14*.

[13] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "Mp-hula: Multipath transport aware load balancing using programmable data planes," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*. ACM, 2018, pp. 7–13.

[14] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 10.

[15] A. Gember, A. Akella, A. Anand, T. Benson, and R. Grandl, "Stratos: Virtual middleboxes as first-class entities," 2012.

[16] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proceedings of the 1st acm sigcomm symposium on software defined networking research*. ACM, 2015, p. 14.

[17] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*.

[18] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 217–230, 2012.

[19] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," *arXiv preprint arXiv:1806.04385*, 2018.

[20] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven network programming," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 369–385.

[21] K. Gao, T. Nojima, and Y. R. Yang, "Trident: toward a unified sdn programming framework with automatic updates," in *ACM SIGCOMM'18*.

[22] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for netkat," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 328–341, 2015.