Enhancing Control Consistency Of Software-Defined Networking

Wen Wang

Doctor of Philosophy

School of Computer Science

McGill University

Montreal, Quebec

August, 2017

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

To my beloved parents.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Prof. Wenbo He, for her patience, guidance and encouragement in the past four years. From her, I learnt how to conduct solid research work, how to present to the public as well as how to collaborate with people. She always supported me to explore anything I would like to try, and guided me to find the right directions when I met problems. Besides the training on research, Prof. He also offered me with a lot of opportunities to attend conferences which led me to world-class research. I also would like to thank my co-supervisor Prof. Xiao-wen Chang for his advice and promotions for continued advancement towards graduation.

I want to thank Prof. Jinshu Su for collaboration in various research projects. His keen insights and rich experience made our collaboration an enjoyable and rewarding experience. I also would like to thank Dr. Cong Liu, who is always patient and helpful, for his valuable feedback and suggestions to my research and thesis. I enjoyed my research time at McGill, and I thank my labmates Nan Zhu, Yixin Chen, Xinye Lin, Joseph D'silva, Sanjay Thakur, etc., who helped me a lot in both research and daily life. I am grateful to Prof. Bettina Kemme accommodated me in her lab in the last year and provided me invaluable advice for my research. I want to thank Prof. Muthucumaru Maheswaran and Prof. Yusheng Ji to serve as my dissertation examiners. I also would like to thank Prof. Mark Coates, Prof. Bettina Kemme, Prof. Muthucumaru Maheswaran, Prof. Hamed Hatami and Prof. Kai-Florian Storch to serve on my committee and provide feedback on the dissertation.

Above all, I would like to thank my parents for their enduring love and support. I dedicate this dissertation to them.

ABSTRACT

As an emerging paradigm in recent years, Software-Defined Networking (SDN) promises flexible programmability and simplified management of networks, and it has gained considerable momentum in both academia and industry. Although the decoupling of the control plane and data plane contributes to the significant benefits of SDN, it also creates challenges in controlling the network consistently and efficiently. First, the control plane manipulates flow tables in the data plane frequently, but the timescales of applying updates in switches vary, such that inappropriate update order would lead to incorrect processing behaviors. Second, with various control programs from different domains running simultaneously to control the network, it is inevitable that control programs may make contradictory decisions, which could cause chaos in configuring the network without reconciliation. Third, as SDN is still in a relatively early stage, the upgrade from a traditional network to a full SDN deployment is usually an incremental process, which necessitates the centralized SDN control and the distributed traditional network protocols working in the same network harmoniously with considerable coordination.

A correct and consistent SDN control is essential to ensure the effective and efficient network behaviors. However, the consistency maintenance always requires extra overhead by introducing further checking, which delays the reactions to network events. Moreover, the consistency requirements restrict the flexibility of SDN by prohibiting some control decisions with potential high performance but violating consistency properties.

In this dissertation, we propose systematic approaches to enhance the control consistency and reduce the overhead and limitations of consistency maintenance.

1) To maintain a consistent view among flow tables during data plane updating, we schedule a fast and efficient update order for forwarding path updates while preserving throughputs of flows. We design a divide-and-conquer approach to overcome the long ordering latency in prior methods and improve the update parallelism.

2) We propose a control coordination approach with declarative languages to compose control programs and reconcile conflicts conveniently. It guarantees the effectiveness of generated control decisions and maximizes the control utility by satisfying the maximum control objectives.

3) To handle the heterogeneity of network devices in hybrid SDN, we design a series of mechanisms to adapt the centralized SDN control to the remaining traditional networking technology, ranging from the placement planning of SDN switches to hybrid traffic engineering. These mechanisms not only coordinate the SDN control with traditional network protocols consistently, but also exert the benefits of SDN.

Together these systems propose a consistent SDN control platform. This control platform has the properties of fast data plane updating, elegant conflicts resolution and effective hybrid control to promise correct enforcement of control policies.

ABRÉGÉ

En tant que paradigme émergent au cours des dernières années, Networking Défini par Logiciel (SDN) promet une programmation flexible et une gestion simplifiée des réseaux, et elle a gagné un élan considérable dans les milieux universitaires et l'industrie. Bien que le découplage du plan de contrôle et du plan de données contribue aux avantages importants de SDN, il crée également des défis dans le contrôle du réseau de manière cohérente et efficace. Tout d'abord, le plan de contrôle manipule fréquemment les tables de flux dans le plan de données, mais les délais d'application des mises à jour dans les commutateurs varient, de sorte que l'ordre de mise à jour inapproprié entraînerait des comportements de traitement incorrects. Deuxièmement, avec divers programmes de contrôle de différents domaines exécutés simultanément pour contrôler le réseau, il est inévitable que les programmes de contrôle puissent prendre des décisions contradictoires, ce qui pourrait provoquer un chaos dans la configuration du réseau sans rapprochement. Troisièmement, comme SDN est toujours dans un stade relativement précoce, la mise à niveau d'un réseau traditionnel vers un déploiement complet de SDN est habituellement un processus incrémental, ce qui nécessite le contrôle SDN centralisé et les protocoles de réseau traditionnels distribués travaillant dans le même réseau harmonieusement avec une coordination considérable.

Un contrôle SDN correct et constant est essentiel pour assurer des comportements de réseau efficaces et efficients. Cependant, la maintenance de la cohérence nécessite toujours des frais généraux supplémentaires en introduisant une vérification supplémentaire, ce qui retarde les réactions aux événements du réseau. En outre, les exigences de cohérence limitent la flexibilité de SDN en interdisant certaines décisions de contrôle avec des performances élevées potentielles mais violant les propriétés de cohérence.

Dans cette dissertation, nous proposons des approches systématiques pour améliorer la cohérence du contrôle et réduire les frais généraux et les limites de la maintenance de la cohérence.

1) Pour maintenir une vue cohérente entre les tables de flux lors de la mise à jour, nous organisons une commande de mise à jour rapide et efficace pour les mises à jour des chemins de routage. Nous concevons une approche de partage et de conquête pour surmonter la longue latence de commande dans les méthodes antérieures et améliorer le parallélisme de mise à jour.

2) Nous proposons une approche de coordination de contrôle avec des langages déclaratifs pour composer des programmes de contrôle et concilier facilement les conflits. Il garantit l'efficacité des décisions de contrôle générées et maximise l'utilité de contrôle en satisfaisant les objectifs de contrôle maximum.

3) Pour gérer l'hétérogénéité des périphériques réseau dans le SDN hybride, nous concevons une série de mécanismes pour adapter le contrôle SDN centralisé à la technologie de réseau traditionnelle restante, allant de la planification de placement des commutateurs SDN à l'ingénierie du trafic hybride. Ces mécanismes ne collaborent pas seulement avec le protocole SDN avec les protocoles réseau traditionnels, mais aussi exercer les avantages de la SDN.

Ensemble, ces systèmes proposent une plate-forme de contrôle SDN cohérente. Cette plate-forme de contrôle possède les propriétés d'une mise à jour rapide du plan de données, d'une résolution de conflits élégante et d'un contrôle hybride efficace afin de garantir une application efficace des politiques de contrôle.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Computer networks typically interconnect a large number of network devices such as routers, switches and numerous types of middleboxes. With more and more business and enterprise network requirements in enhancing connectivity and performance, networking protocols have evolved significantly over the last few decades. In this chapter, we will describe the features of traditional networking, and the need for a more flexible network management. The birth of Software-Defined Networking (SDN) creates a new perspective to overcome the ossification of traditional networking for network innovations. However, SDN also faces challenges to ensure the correctness and efficiency with the new networking architecture. We describe the contributions to address these challenges in this dissertation.

1.1 Current Networking Paradigms

1.1.1 Traditional Networking

Traditional networks consist of a collection of devices which are running various network protocols to process packets. Each network device is composed of the control plane and data plane as Figure 1.1 shows. The control plane makes forwarding decisions and instructs the data plane to enforce policies in the forwarding table. The data plane processes packets at high speed using fast on-chip memory which stores packet-processing rules, such as filtering, forwarding, buffering, scheduling, etc. Each device plays an equal role in the network, and contributes substantially

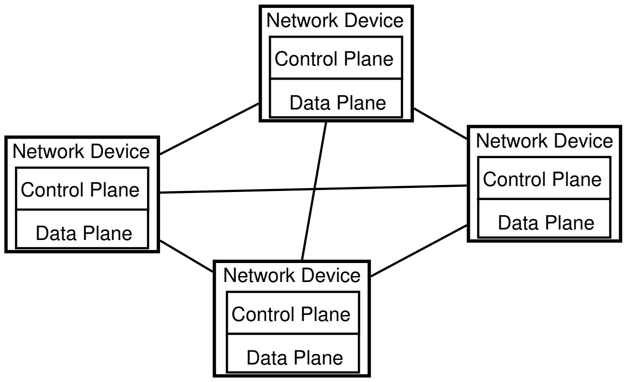


Figure 1.1: Traditional networking

to the robustness of the traditional networking systems. A single node failure would not impact the rest of the network beyond the loss of connectivity caused by that failure. However, the widely applied traditional networking architecture is ossified, which hinders the innovation of network algorithms in the past over 20 years.

Difficulty of new protocol design: Network switches process packets based on the layers defined by the network protocol stacks, e.g., TCP/IP, UDP/IP. The design and development of new network protocols and mechanisms in traditional networks also follow the protocol stacks. Because of the layered architecture, designing an effective protocol usually involves information of multiple layers, so that modifications to a layer inevitably require the awareness or even changes in other layers, especially with cross-layer optimizations [1, 2]. Moreover, as the control plane is distributed in each device, it can not make network-wide decisions effectively without a global view of the network. This further adds to the difficulty of designing new control algorithms. Changing the basic nature of IP, TCP, DNS, or the Sockets API is quite difficult [3]. Therefore, Internet protocol stack has become ossified to an hourglass-shaped architecture [4, 5].

Difficulty of new algorithm deployment: The control plane is tightly coupled with the data plane in traditional networks. The control functionality is implemented within each device, and most functionality is implemented in dedicated hardware, e.g., Application Specific Integrated Circuit (ASIC). Due to the distributed control, introducing new functionality to the network requires deploying the new functionality in each device independently with a heavy labor cost, or even hardware replacement with a large economic cost. These costs impede the fast and wide deployment of innovative algorithms. Even the transition from IPv4 to IPv6 started more than a decade ago is still largely incomplete [6].

Difficulty of device configuration: A network is usually composed of a variety of devices from different vendors. Devices of different vendors usually equip with different, closed and proprietary implementations and interfaces. To successfully complete a configuration, an administrator needs extensive knowledge of all device types. Therefore, it requires a high level of expertise. Moreover, manually translating high-level policies into low-level vendor specific configurations is a time-cosuming and error-proning task due to the complex dependencies among policies. Therefore, device configuration can be a challenging process for complex networks such as data center networks and enterprise networks.

These characteristics of traditional networking make the management and configuration of networks difficult, and hinder the evolving of network protocols. The architecture ossification leads to overly complex and inflexible networks, which is hard to support today's growing network traffic and performance requirements.

1.1.2 Software-Defined Networking

Software-Defined Networking (SDN) is an emerging networking paradigm to control network in recent years. It promises to simplify network management and enable

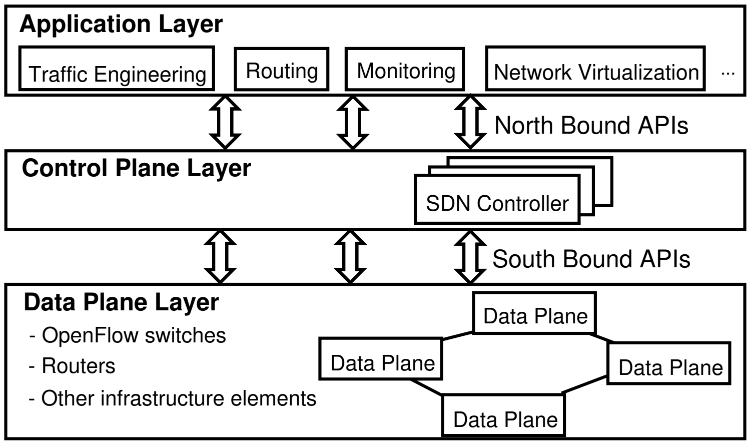


Figure 1.2: SDN architecture

innovation and evolution. SDN could be abstracted as three layers in Figure 1.2: application layer, control plane layer and data plane layer. SDN separates the control functionality from data plane, and the control plane bridges the application layer and data plane with well-defined north bound and south bound APIs.

- Application layer: SDN applications are control programs that communicate their network requirements and desired network behaviors to the control plane via a northbound API. These applications include networking management, measurement, analytics, etc.
- Control plane layer: The control plane layer usually runs as a SDN controller which is a logically centralized entity. The controller receives instructions or requirements from the application layer and relays them to the network devices. It also extracts information about the devices and communicates back to the SDN control programs with an abstract view of the network, including network events and statistics.
- Data plane layer: The data plane layer consists of network devices that expose their visibility and control to the controller. As control functionality is removed

from network devices, SDN switches are pure forwarding devices and process packets based on the instructions received from the control plane.

SDN eliminates the complex and static nature of the traditional networking architecture, and it offers more programmability and fine-grained control for the network.

Decoupled control plane and data plane: SDN decouples the control plane from the data plane, and the control plane is a logically centralized controller. Novel network control algorithms could be developed and integrated in the controller conveniently and control the network programmatically, which overcomes the deployment difficulty of innovative protocols in traditional networks. The controller gathers information from the data plane and has a global view of the network. Thus, control programs running on top of the controller could make network-wide optimized control decisions with the global information, which used to be a design difficulty of effective protocols in traditional networks. This change promises to greatly accelerate the pace of innovation in networks [7].

Standardized device configuration: The control plane is responsible for translating requirements of control programs into policies which are to be applied in the data plane. The controller configures the data plane with remote communication. OpenFlow [8, 9] defines a standard configuration API, so that devices of different types and vendors are configured in a generic and convenient manner, which resolves the device configuration difficulty in traditional network. An OpenFlow switch initially establishes a communication channel over a TCP connection with a controller, and then exchanges information with the controller to allow it to configure the switch's flow table. The controller is able to add, modify or delete rule entries in flow tables with control messages through the channel, which essentially controls the processing behaviors of the data plane.

Generalized packet processing: Switches process packets based on the rules in flow tables, which are installed by the controller. Each rule entry is in the format of match-action. The match field specifies the header pattern of packets, and the action field specifies the processing action for matched packets. Upon the arrival of a packet at an OpenFlow switch, if a matching entry is found in the flow table, the switch applies the corresponding action to the packet. Otherwise, a message is sent to the controller requesting the processing action. The match-action abstraction is general enough to support a variety of functions beyond simple L2 or L3 forwarding, e.g., access control rules and traffic monitoring [10, 7]. The generalization of match-action flow table makes the configuration of network devices simplified. It also drives the development of the network device hardware of various vendors, e.g., Cisco, HP, Arista, BigSwitch, etc.

The separation of control functionality and underlying forwarding is the key to the flexibility of SDN, which breaks the control problem into tractable pieces. Therefore, it is easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution and innovation [11]. While SDN was originally considered in the context of campus networks and data centers, it gains significant attractions in the industry over the past few years, e.g., Google [12, 13], Facebook, Yahoo, Microsoft [14, 15], and is being considered in Wide Area Networks (WANs) [16, 17], carrier networks, and mobile backhaul networks [18] for traffic engineering, network monitoring, network virtualization, etc.

1.2 Challenges Of Control Consistency In Software-Defined Networking

While the decoupling of control plane and data plane makes SDN more flexible and programmable, it also creates a lot of challenges to maintain network processing correctness and control efficiency due to the remote control [19, 20], which do not exist in traditional networks. To ensure the correctness and performance of SDN, in this dissertation, we focus on the consistency and efficiency of SDN control.

1.2.1 Control Consistency In SDN

Data plane update consistency: As the control plane regulates the network states by manipulating flow tables in data plane, flow tables should be updated timely to react to network events correctly and efficiently. Today, SDN is widely applied for load balancing and failure recovery. In these applications, the controller schedules flows to redundant paths to avoid network congestions and failures by updating flow tables in switches with control messages. Flow table updates may take effect in a delayed and asynchronous manner: not only because updates have to be transmitted from the controller to the switches over the network, but also the reaction time of the switches to updates may differ, depending on hardware or concurrent load [21]. However, inconsistency may occur if a switch is updated before another improperly. Due to the coexistence of old rules and new rules in switches, inconsistent flow table updating may lead to transient incorrect network behaviors (e.g., loops, black-holes) or undesired performance degradation (e.g., transient communication interruption). For example, a flow is forwarded to a switch A which has not been updated, while the old rules for this flow on other switches have been removed. The flow still follows the old rule on A which probably leads it into an undesired forwarding path with incorrect matching or no matching on downstream switches, and no matching would initiate a request to the controller for the forwarding rule with some processing delay. Thus, lack of consistency during the data plane update could adversely impact the stability and availability of the network. Although flow tables are distributed in switches, the manipulations made by the control plane should ensure flow tables have a consistent view at any time to guarantee the forwarding correctness and availability without any loops or black-holes.

Control plane consistency: With the increasing applications of SDN in real-world networks, especially enterprise network, more and more SDN control programs from multi-domains are running simultaneously to configure a network. It is inevitable that control programs make conflicting control decisions, e.g., contradictory forwarding actions. The control decisions made by different control programs must be reconciled to be consistent before applying in the data plane. Otherwise, inconsistent control decisions would lead to misconfiguration or performance degradation in the network. For instance, a firewall control program would block a flow, while the routing control program has already calculated the forwarding path for it. With the contradictory decisions, only one decision should be applied in the data plane, in which switches simply process packets according to the matched rules with the highest priority. It is the controller's responsibility to decide which policy to take effect to ensure the correctness and concision of the data plane. Moreover, to relieve the performance bottleneck on the control plane, the control plane is usually logically centralized with multiple control nodes. Maintaining a consistent global view among all the controllers is essential to ensure desired network operations.

Consistency in hybrid SDN: The upgrade of a legacy network to a full SDN deployment is usually an incremental process, during which SDN switches controlled by the controller and legacy switches running traditional network protocols coexist in the hybrid network. However, distributed protocols in legacy switches are out of SDN control, which probably restricts the flexibility of SDN. For instance, the forwarding on legacy switches is decided by distributed routing protocols, while SDN switches could forward packets to their neighbours flexibly based on instructions from the controller. The controller has to avoid generating contradictory forwarding decisions with distributed routing, e.g., leading packets into a loop without the awareness of distributed routing decisions, otherwise the network will be in a chaos. Therefore, the hybrid SDN requires considerable coordination between the centralized control and traditional network protocols to maintain the consistency.

Considering these consistency requirements, an inappropriate design of control programs violating any consistency could not exert the advantages of SDN, and it even results in performance degradation or erroneous network behaviors.

1.2.2 Efficiency And Effectiveness Of Consistent SDN Control

Consistency maintenance overhead: The consistency maintenance usually imposes further checking and coordination of SDN control decisions before applied in the data plane, e.g., update sequence ordering [22, 23], locks on variables [24], which introduces extra overhead and control delay. An efficient and consistent SDN controller should have low overhead and short latency to react to network events quickly [25]. Moreover, the controller may install multiple rules of different versions in switches to maintain the consistency during data plane updating, e.g., two-phase update [26], which adds overhead to the limited flow table space. Although the consistent network update problem is not introduced by SDN, the decoupling of control plane and data plane as well as the flexibility of SDN are likely to increase the frequency of data plane update, e.g., for supporting more fine-grained and frequent optimization of forwarding paths [21]. Therefore, maintaining the consistency efficiently is a great challenge for SDN.

Restricted SDN performance and flexibility: The dependencies among rules imposed by consistency (e.g., one rule must update before another) fundamentally limit how quickly the data plane can be updated, e.g., updating rules according to the calculated sequence [22, 23], which adds to the delay of network processing. Furthermore, the consistency requirements limit the potential flexibility of SDN, e.g., restricted SDN forwarding in hybrid SDN to avoid conflicting with traditional routing. Due to the limitation, the SDN controller has to avoid a lot of control policies with potential high performance but not complying with consistency requirements, which constricts the desired network performance. Therefore, there is a trade-off between the consistency and performance. An effective and efficient SDN control program should achieve high network performance utilizing the benefits of SDN while without violating the consistency.

Enforcing policies consistently and efficiently across the network is a basic requirement for SDN [27]. Therefore, SDN requires considerate control designs to ensure the correctness and performance.

1.3 Contribution

The motivation for this dissertation is to deal with the consistency concerns in Section 1.2 efficiently and effectively, and we develop control approaches to maintain the control consistency in three aspects. Figure 1.3 shows an overview of the contribution architecture. First, we design a fast and efficient update ordering algorithm to ensure consistent and congestion-free update in data plane (Chapter 2). Second, we propose an approach to coordinate control programs to generate consistent policies and reconcile control conflicts (Chapter 3). Third, we design SDN control to exert the benefits of partially deployed SDN in hybrid SDN, while complying with the traditional networking features (Chapter 4). These components are deployed in the

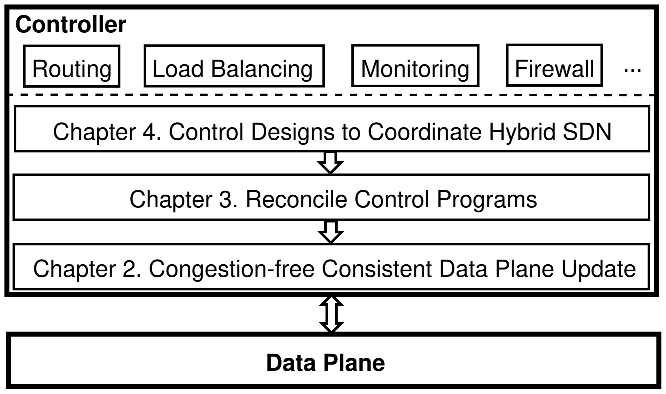

Controller
Routing
Load Balancing
Monitoring
Firewall
...
Chapter 4. Control Designs to Coordinate Hybrid SDN
Chapter 3. Reconcile Control Programs
Chapter 2. Congestion-free Consistent Data Plane Update
Data Plane

Figure 1.3: Overview of contributions to SDN control consistency

control plane, and each component interacts with another: the data plane update component schedules the update order for the reconciled control policies from the control program reconciliation component, and the reconciliation component checks decisions made by the hybrid coordination component for consistency, forming a bottom-up design.

Consistent and congestion-free data plane update (Chapter 2): The consistency among flow tables is critical to ensure correct network behaviors during data plane updating. As the consistency requirements impose dependencies among rules in flow tables, the order of updates must be carefully considered. To update flow tables consistently and efficiently, we propose an efficient and effective update ordering approach – Cupid. To avoid large overhead in update ordering, we divide the global dependencies among updates into local restrictions, which is the key to reduce the dependency complexity and overcome the long ordering latency of prior approaches. We design a heuristic algorithm in resolving the dependency to calculate the update order, which promises a fast and parallel update in the data plane. This part of the work is published in IEEE International Conference on Computer Communications (INFOCOM 2016) [23].

Reconciling SDN control programs (Chapter 3): Existing SDN control coordination approaches either compose control programs to derive consistent decisions jointly or examine the decisions of each control program to ensure the consistency. However, the former is usually of great complexity and hard to be conducted automatically, and the latter probably results in suboptimal solutions due to the independent execution of control programs. Moreover, these approaches all fail to maximize the utility of the control programs. To reconcile the SDN control, we propose Redactor to optimize the consistency and utility of network control in an automatic and dynamic manner. In order to make network control consistent, we implement SDN control programs with declarative language Prolog, and compose control programs automatically to execute together to make consistent decisions. When conflicts occur, we use a heuristic approach to compromise a subset of control programs to maximize the control utility. This part of the work is published in IEEE International Conference on Network Protocols (ICNP 2016) [28].

Boosting the benefits of hybrid SDN (Chapter 4): The long incremental upgrade process from a traditional network to a full SDN deployment requires coordination between the centralized SDN control and traditional distributed network protocols to guarantee correct behaviors of the hybrid network during upgrading. To ensure the control consistency and manage the network efficiently, we design considerate SDN control to handle the heterogeneity caused by distinct forwarding characteristics of SDN and legacy switches, aiming to boost the benefits of hybrid SDN. Our solutions range from the placement of SDN switches in the hybrid network to the hybrid traffic engineering. We consider the features of both the centralized SDN control and traditional networking in the SDN control design. Therefore, the benefits of the partially deployed SDN are achieved while the consistency between the SDN control and traditional networking maintains. The preliminary results have been published in IEEE International Conference on Distributed Computing Systems (ICDCS 2017) as a short paper [29], and the full paper is in submission and under review.

CHAPTER 2
Cupid: Congestion-free Consistent Data Plane Update

This chapter focuses on the consistency during updating forwarding rules in the data plane. When the forwarding paths of flows change for load balancing or failure recovery purpose, the controller updates rules in switches with control messages for these flows. The consistency requirements impose dependencies among updates of rules (e.g., one rule must update before another) in different switches across the network. Therefore, finding a correct and efficient update order is critical to ensure the desired network behavior. Previous update methods usually check the global dependency among updates dynamically to schedule an update order, in which the complex dependencies make the scheduling slow and can not adapt to runtime networks. In this chapter, we divide the global dependency into local dependencies to reduce the dependency complexity and improve the scheduling parallelism, which finally speeds up the update scheduling.

2.1 Introduction

Software-Defined Networking has been widely applied for traffic engineering [16, 30] and failure recovery [31] with its global view. The SDN controller schedules flows to other available paths and updates flow tables in concerned switches for load balancing or failure recovery. Although plenty of researches [32, 33, 34, 35, 36] have been carried out to compute optimized routing paths based on current network topology and traffic distribution to protect against failures and congestions, a common challenge faced in all centrally-controlled networks is updating the data plane consistently and efficiently [22].

The consistency requires that flows are migrated to new routing paths seamlessly, never with loops nor black-holes during flow tables updating, which imposes dependencies among rules in flow tables along a routing path [37]. The missing or mismatch of forwarding rules in switches may interrupt a flow for a while. Therefore, the order of updating flow tables in concerned switches must be carefully considered. Moreover, in networks where communications usually have sensitive performance requirements, such as data centers, a stronger consistency instructs that traffic should not exceed link bandwidth capacity during updating, i.e., congestion-free consistency [17]. With limited bandwidth resource of links, even though the bandwidth demands of flows are satisfiable before and after data plane updating, flows may be rerouted to a link before offloading original flows on the link, which may result in congestions and throughput degradation during updating. Therefore, the congestion-free consistency further poses more dependencies among updates.

To update flows to new routing paths without any performance degradation, [17, 38] formulates the problem as LP (Linear Program) to find a transition sequence from the initial state to the target state. However, this approach would be quite slow and does not scale to large networks with a large number of flows. Heuristic approaches trying to find an updating order to resolve dependencies among updates, e.g., [22, 39], also suffer substantial overhead due to the high dependency complexity. Meanwhile, to ensure the loop free and black-hole free consistency of each flow, two-phase update [26] is proposed which forwards packets either with the new path or the old path, but never with a mixed path. Unfortunately, the atomic commit adds to the complexity of dependencies among updates. As congestions may occur on any hop of a new routing path during updating, to avoid throughput degradation, a flow only could migrate to the new path until all these hops have enough available bandwidth, which brings down updating efficiency. Moreover, a flow table may hold multiple entries of different versions for each flow in two-phase update. This may overload the limited flow table space. Attempts to reduce flow table space overhead during updating have been made in [40, 41, 42], but they always have to make trade-offs between flow table space and updating efficiency.

In this chapter, we present a congestion-free update ordering approach while maintaining the black-hole free, loop free consistency properties. We firstly find that the new routing path of each flow could be divided into several independent segments, and identify the critical nodes which control traffic shifting between the old path and new path. To avoid congestions during updating, instead of resolving a global dependency graph composed of updates and network resources in [22], we divide dependencies among updates into local dependencies among critical nodes, and then construct a dependency graph with potential congested links. The divided local dependencies improve update parallelism, and ensure the efficiency and scalability of congestion-free updating. In this way, we successfully restrict the problem space while keeping the dependency. We then design and implement a heuristic dependency resolution algorithm to schedule a fast data plane update order. Meanwhile, to reduce the flow table space overhead, we use multiple input and output ports with weights in each flow entry, so that a switch keeps only one rule for each flow during updating. The results of simulation show that Cupid schedules update ordering at least 2 times faster than Dionysus [22] and has less throughput losses in both fat-tree and mesh topologies.

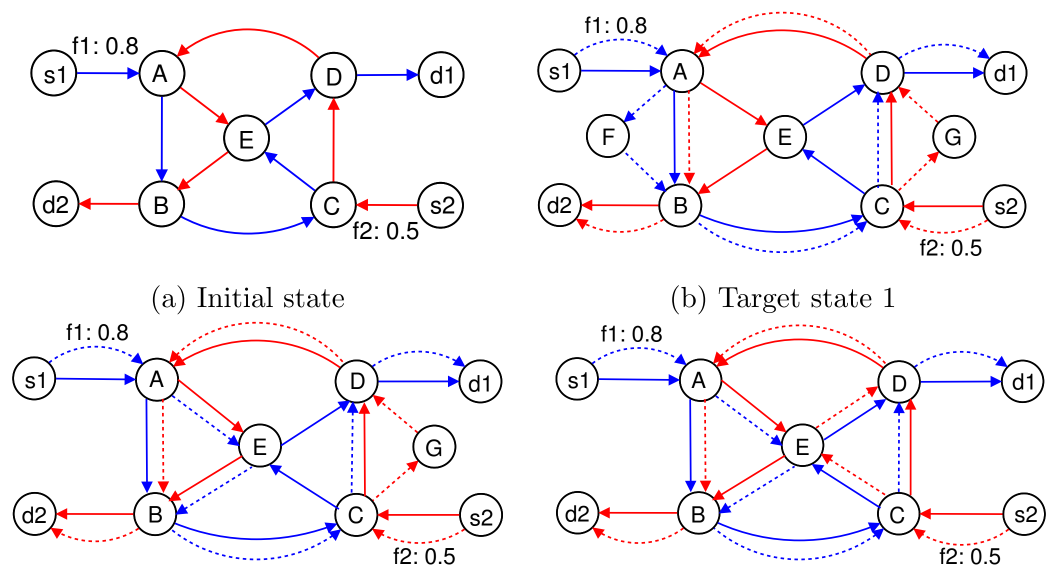
2.2 Challenges And Related Work

2.2.1 Complexity Of Congestion-free Updating

To achieve high network utilization during flow tables updating, [17, 38] formulates the updating problem as a LP problem to find a transition sequence from the initial state to the target state for inter-data center WANs and inside data center networks respectively. However, real-world networks are usually more complicated than the well organized inter- and intra- data center networks. A more complex network tends to involve more complicated dependency among updates and even dependency deadlocks which complicate the update ordering. In Figure 2.1a, flows f_1 and f_2 are rerouted to other paths to release more available bandwidth for future flows on links $A \to B$, $C \to E$ and $E \to D$. With different rerouting schemes, the target routing paths (dotted lines) are different in Figure 2.1b, 2.1c, 2.1d, which result in differences in update ordering. Moreover, the three updating scenarios vary in dependency complexity.

In Figure 2.1b, when flow f_1 is rerouted from subpath $C \to E \to D$ to $C \to D$, it has to wait for the removal of f_2 on link $C \to D$ due to the bandwidth limitation. Meanwhile, the rerouting of f_2 from subpath $A \to E \to B$ to $A \to B$ depends on the remove of f_1 on link $A \to B$. As f_1 could update from $A \to B$ to $A \to F \to B$, and f_2 could also shift from $C \to D$ to $C \to G \to D$ freely without any bandwidth restriction, the dependencies are released. Thus, we can schedule a feasible update order to solve the dependencies.

In Figure 2.1c, when f_1 updates from subpath $A \to B$ to $A \to E \to B$, it requires moving f_2 from $A \to E \to B$ to $A \to B$ firstly. However, due to the bandwidth restriction, it is impossible to make the exchange without any performance reduction with single path forwarding. Therefore, [22, 17, 38] use multipath to migrate flows



(a) Initial state

(b) Target state 1

(c) Target state 2

(d) Target state 3

Figure 2.1: Routing update of flows: (a) shows the current routing paths (solid lines) of flows f_1 (blue from s_1 to d_1) and f_2 (red from s_2 to d_2), and then f_1 and f_2 may be rerouted to new paths (dotted lines) in (b), (c), (d) respectively with three different routing schemes. The bandwidth capacity of each link is 1 unit, and throughputs of f_1 and f_2 are 0.8 unit and 0.5 unit.

gradually to new paths, e.g., splitting f_1 into 0.5 unit and 0.3 unit on subpaths $A \to E \to B$ and $A \to B$ respectively, thus f_2 could shift to $A \to B$ completely with 0.5 unit, and then f_1 finally migrates 0.3 unit on the old subpath to $A \to E \to B$. Unfortunately, in this scenario, if we split f_1 into two subflows with $A \to B$ and $A \to E \to B$ before updating the rule in C from $C \to E$ to $C \to D$, we will get a loop $\{E, B, C\}$ on $f_1's$ routing path.

Despite of the multipath transition with $A \to B$ and $A \to E \to B$, Figure 2.1d also requires multipath transition with $C \to D$ and $C \to E \to D$ for f_1 and f_2. Furthermore, to avoid loop $\{E, B, C\}$, the removal of rule $C \to E$ for f_1 in switch C requires the multipath transition of f_1 and f_2 with $C \to D$ and $C \to E \to D$, which further requires the removal of rule $A \rightarrow E$ for f_2 in switch A to avoid loop $\{E, D, A\}$, so that a deadlock occurs between the two multipath transitions.

2.2.2 Efficiency Of Update Ordering

Although congestion-free updating is usually complicated in real-world networks, updates should react in time to routing changes to minimize the duration of performance degradation and network failures [43]. However, maintaining congestion-free consistency during updating usually requires global coordination due to complex dependencies among updates, which poses challenges to update ordering efficiency. LP [17, 38] would be too slow to find a feasible ordering with the complex dependency. Dionysus [22] proves that finding the fastest update scheduling is a NP-complete problem, and dynamically schedules a dependency graph among updates and network resources with a heuristic algorithm. However, with the complicated topology and strong dependencies in real-world networks, the global dependency graph coordination is of great overhead. Hong et al. [17] and Shi et al. [44] also note that careful ordering of updates cannot always guarantee congestion freedom during updating. Zhou et al. [43] and McClurg et al. [45] reduce the consistency problem to a model checking problem instead of designing a new ordering algorithm.

In spite of the low efficiency due to the global coordination overhead, the consistency in two-phase update [26] requires forwarding packet either with the new or the old path, but never with a mixed path, so that all the switches in a new path must be updated before shifting any traffic of a flow to the new path. Thus, the strong property of consistency would lead to a long delay for rerouting update. Moreover, we must make sure neither the new path nor the old path is congested, while any hop on a routing path may be exposed to congestions during updating. Ordering and two-phase approaches both could benefit from time-triggered updates [46, 42] but with extra clock synchronization in switches. To improve the update efficiency, [37, 47] note that dependency structures are simpler for weaker consistency properties than stronger properties, and make a trade-off between the strength of consistency property and dependencies.

2.3 Independent Segments Of A Single Flow

2.3.1 Consistency With Mixed Path

Considering the low efficiency of the strong consistency, we note that the black-hole free and loop free consistency could still be preserved with mixed paths without the atomic committing in two-phase update. For instance, in Figure 2.1b, a packet of flow f_1 going through $s_1 \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow d_1$ or $s_1 \rightarrow A \rightarrow F \rightarrow B \rightarrow C \rightarrow E \rightarrow D \rightarrow d_1$ with mixed subpaths still ensures the connectivity of f_1. Zhou et al. [43] notes that the black-hole free and loop free consistency is a downstream-dependent property. Updating from downstream switches to upstream switches sequentially is sufficient to ensure the consistency, and we call it reverse order updating in Lemma 1.

Lemma 1. *The reverse order updating of a routing path is black-hole free and loop free.*

However, upstream switches always have to wait for the update completion of downstream switches in reverse order updating. Actually, some upstream switches are independent from downstream updates as long as connectivity maintains. In Figure 2.1b, switch A does not need to know the downstream path of C is $C \rightarrow E \rightarrow D \rightarrow d_1$ or $C \rightarrow D \rightarrow d_1$ for flow f_1 provided the connectivity between C and d_1. Similarly, C also does not care about the upstream path of flow f_1 is $s_1 \rightarrow A \rightarrow F \rightarrow B$ or $s_1 \rightarrow A \rightarrow B$ as long as packets arrive at C correctly. In other words, the updates of flow f_1 to subpaths $A \rightarrow F \rightarrow B$ and $C \rightarrow D$ are independent.

To understand how switches migrate a flow from its old routing path to a new routing path, we identify the critical nodes which control the routing path switching. The critical nodes $C(f)$ of a flow f are the common switches on both the old path $P_o(f)$ and the new path $P_n(f)$, but using different rules.

$$C(f) = \{n_f | n_f \in P_o(f) \cap P_n(f), rule(n_f, P_o(f)) \neq rule(n_f, P_n(f))\}$$

As critical nodes connect the new path with the old path, such as switches A, E, B on f_1's routing paths in Figure 2.1c, these critical nodes control the traffic shifting between the old path and new path. Therefore, the updates of critical nodes are of great importance during updating. Based on the modifications made to flow entries, critical nodes could be divided into three classes: $(inport, output)$ in which both the input and output ports require to be modified (e.g., E for f_1), $(*, output)$ changes the output port of flow entry while the input port stays unchanged (e.g., A for f_1), and $(inport, *)$ only changes the input port (e.g., B for f_1).

To save flow table space, each flow entry is equipped with multiple input or output ports in updating critical nodes. Thus, there is only one flow entry kept for each flow in a switch. Especially for the critical nodes which require output port modifications, these nodes control how to forward packets through mixed paths with multiple outports. Each outport is associated with a weight, so that these weights determine the amount of traffic on the new path and old path to avoid congestions during updating.

2.3.2 Segment Partition Of A flow

To make the updating flexible and reduce the updating latency, we divide the new routing path of flow f into several segments $Sg(f) = \{sg\}$, and each segment sg could be updated independently. Each segment sg consists of a minimum sequence of switches on the new path which starts and ends with critical nodes (or the source and destination switches for the first and last segments), so that the beginning and ending nodes control traffic shifting between the old subpaths and new segments. Adjacent segments only share the ending or beginning nodes and never overlap on nodes inside segments. Thus, each segment is an atomic sequence, and can be represented as:

$$\begin{aligned} Sg(f) =&\{sg \subseteq P_n(f) | \min\{sg.length\}, sg.start, sg.end \in C(f) \cup \{P_n(f).start, P_n(f).end\}, \\ &\forall sg' \in S(f), sg \cap sg' = \varnothing \textit{ or } = sg.start \equiv sg'.end \textit{ or } = sg.end \equiv sg'.start, \\ &lasthop(sg) \notin cycle(P_o(f), P_n(f))\} \end{aligned}$$

As the beginning and ending nodes of each segment are critical nodes, they always need to update either input port or output port. Even though a beginning node may require updating the input port field, to make each segment independent, the updating of the beginning node in a segment only modifies the output port field $(*, outport)$ of the flow entry, while the modification of the input port $(inport, *)$ belongs to the ending node of last segment. Similarly, the ending node of each segment only modifies the input port field $(inport, *)$, while the update of output port $(*, outport)$ belongs to the beginning node of the next segment.

To avoid loops during updating, the last hop of a segment should not belong to any cycle formed by the old path and new path with $lasthop(sg) \notin cycle(P_o(f), P_n(f))$. If there is a segment of flow f_1 in Figure 2.1c ending with $\rightarrow E \rightarrow B \rightarrow C$ which is involved in the cycle $\{E, B, C\}$, packets will be circulated if we update the routing path to $E \rightarrow B \rightarrow C$ in the segment before removing the rule $C \rightarrow E$ in C which belongs to the next segment. Hence, the cycle makes the updating of $E \rightarrow B \rightarrow C$ depends on updating f_1 to $C \rightarrow D$. Thus, $C \rightarrow D$ should be added to the end of

Algorithm 1 Segment Partition

```
1:  $Sg(f) = \varnothing$
2:  for $n : P_n(f)$ do
3:      $sg = n$
4:      $m =$ successor node of $n$ on $P_n(f)$
5:      while $m \neq \varnothing \ \wedge (m \notin C(f)$ or $(n \rightarrow m) \in cycle(P_o(f), P_n(f)))$ do
6:          $sg = sg \rightarrow m$
7:          $n = m,\ m =$ successor node of $n$ on $P_n(f)$
8:      end while
9:      $sg = sg \rightarrow m$
10:     $Sg(f) = Sg(f) \cup sg$
11:     $n = m$
12: end for
```

the segment to break the cycle, otherwise the updating of the segment depends on the next segment.

We design an algorithm to calculate segments set $Sg(f)$ for each flow f in Algorithm 1. Non-critical nodes and the nodes belonging to any cycle are added to a segment until meeting a critical node breaking the cycle in Line 5-8. $cycle(P_o(f), P_n(f))$ is the cycle set formed by the old path and new path, such as cycles $\{E, B, C\}$ formed by f_1's routing paths and $\{E, D, A\}$ formed by f_2's routing paths in Figure 2.1d. By traversing switches along a new path (Line 2), any switch on the new path is assigned to its segment.

2.3.3 Properties Of Segments

Theorem 1. *A segment $sg \in Sg(f)$ is black-hole free and loop free with reverse order updating.*

Proof. As $sg.start, sg.end \in C(f) \cup \{P_n(f).start, P_n(f).end\}$ are the common nodes of the new and old paths, we construct a flow f' from $sg.start$ to $sg.end$ with the same header field of f. The routing update of f' between $sg.start$ and $sg.end$ follows Lemma 1 with reverse order updating, thus the black-hole free and loop free properties preserve in sg. □

Theorem 2. *Segments in $Sg(f)$ are independent from each other, which means updates in a segment do not depend on updates in other segments.*

Proof. If the updating of switch $sg[i]$ in segment sg depends on another segment $sg' \in Sg(f)$, the updating of $sg[i]$ before updates in sg' will result in a loop l or black hole b.

Switch $sg[i](0 \leqslant i < sg.length{-}1)$ updates $rule(sg[i], P_o(f))$ to $rule(sg[i], P_n(f))$ which establishes a path $sg[i] \to sg[i+1] \in sg$. As $sg[i] \notin sg'$, $sg[i] \to sg[i+1] \notin sg'$, which means no path changes in segment sg'.

As no path changes in sg', the loop and black-hole $l, b \notin sg'$. According to Theorem 1, segment sg is black-hole free and loop free with reverse order updating, which conflicts with l, b. Therefore, $sg[i]$ does not depend on any node in sg'. □

With Theorem 1 and 2, each segment acts as an independent routing path with reverse order updating. Thus, segments could be updated in parallel to improve updating parallelism and flexibility. During updating of independent segments, packets of a flow f may be forwarded along a path composed of mixed nodes belonging to $P_o(f)$ or $P_n(f)$, while the connectivity always maintains.

According to Algorithm 1, the segments of flows in Figure 2.1b,2.1c,2.1d are showed in Table 2.1. Although segments start and end with critical nodes, a lot of segments do not need any update due to the divided updating of *inport* and *outport* between beginning and ending nodes on *inport* and *outport* fields. For instance, the segment $s_1 \to A$ of flow f_1 ends with a critical node A in Figure 2.1b. Switch A

Table 2.1: Segments of flows

Figure	Segments of f_1	Segments of f_2
2.1b	$\{s_1 \to A, A \to F \to B, B \to C, C \to D, D \to d_1\}$	$\{s_2 \to C, C \to G \to D, D \to A, A \to B, B \to d_2\}$
2.1c	$\{s_1 \to A, A \to E, E \to B \to C \to D, D \to d_1\}$	$\{s_2 \to C, C \to G \to D, D \to A, A \to B, B \to d_2\}$
2.1d	$\{s_1 \to A, A \to E, E \to B \to C \to D, D \to d_1\}$	$\{s_2 \to C, C \to E, E \to D \to A \to B, B \to d_2\}$

only needs to update the *outport* field for f_1 with the beginning node in the next segment $A \to F \to B$, so that no node requires updating in segment $s_1 \to A$. Therefore, segments requiring no update are shadowed in Table 2.1, such that only partial segments require updates, which further reduces updating overhead.

2.4 Congestion-free Updating Of Multiple Flows

Even though the independent segment partition and reverse order updating improve updating parallelism and efficiency of each flow while ensuring connectivity, topology changes and load balancing usually require rerouting multiple flows to other available forwarding paths in a short time. However, with bandwidth limitation on links, multiple flows may congest on a link during the migration, which results in performance reduction of several flows. To ensure performance of flows during updating, we have to schedule a feasible congestion-free updating order to update flow tables for multiple flows.

2.4.1 Potential Congested Links During Updating

To avoid congestions during updating, we have to discover potential congestions at first. We define the criteria to identify a potential congested link l between switch u and v as follow:

Definition 1 (Potential Congested link $l = u \to v$). *With flows desire to use the link* $F_n(l) = \{f \mid \forall f, l \in P_n(f) - P_o(f)\}$*, flows to be moved away from the link* $F_o(l) = \{f \mid \forall f, l \in P_o(f) - P_n(f)\}$*, and unchanged flows going through this link* $F_u(l) = \{f \mid \forall f, l \in P_n(f) \cap P_o(f)\}$ *during updating, the throughputs of flows* $b(f)$ *on a potential congested link* l *satisfy*

$$\sum_{f_i \in F_o(l) \cup F_u(l)} b(f_i) \leq c(l), \quad \sum_{f_i \in F_n(l) \cup F_u(l)} b(f_i) \leq c(l) \tag{2.1}$$

$$\sum_{f_i \in F_o(l)} b(f_i) + \sum_{f_i \in F_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) > c(l) \tag{2.2}$$

The consumed bandwidth does not exceed the bandwidth capacity $c(l)$ in both the initial and final states (2.1). However, flows in $F_n(l)$ may be scheduled to link l at any time before moving some old flows in $F_o(l)$ away during updating, which exceeds the link bandwidth capacity (2.2). To avoid any throughput degradation, we must find all potential congestion combinations of $F_n(l)$ and $F_o(l)$ on link l. The ordering of updates related to potential congested links must be carefully considered to avoid congestions. With Definition 1, we can find a set of potential congested links CL which contains all the links that may be congested during updating.

Proof. If link l is congested during updating, but $l \notin CL$, there must exist a subset of newly added flows $F'_n(l) \subseteq F_n(l)$ and old flows $F'_o(l) \subseteq F_o(l)$ on link l when the congestion occurs.

$$\sum_{f_i \in F_o(l)} b(f_i) + \sum_{f_i \in F_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) \geq \sum_{f_i \in F'_o(l)} b(f_i) + \sum_{f_i \in F'_n(l)} b(f_i) + \sum_{f_i \in F_u(l)} b(f_i) > c(l)$$

Therefore, l must be a potential congested link. $\square$

2.4.2 Dependency Graph For Congestion-free Updating

Intuitively, if flows in $F_o(l)$ are moved away from the potential congested link l before flows in $F_n(l)$ shifting to l, congestions will be avoided. Unfortunately, flows are usually blocked by multiple potential congested links on the new path. Thus, the rerouting of a flow may depend on updates of several flows, which makes the update scheduling complicated.

As Figure 2.1b shows, the update of f_1 depends on the removal of f_2 on potential congested link $C \to D$, while f_2 depends on the removal of f_1 on potential congested link $A \to B$. The dependencies seem making a deadlock. With segments partition, the updating of f_1 to $C \to D$ falls into segment $(C \to D)_{f_1}$ (we use $(segment)_{flow}$ to indicate a segment of a flow) while the removing of f_2 from $C \to D$ is in segment $(C \to G \to D)_{f_2}$. Similarly, the updating of f_2 to $A \to B$ is in segment $(A \to B)_{f_2}$, while the removing of f_1 falls into $(A \to F \to B)_{f_1}$. Thus, dependencies among flows could be divided into local dependencies among segments, e.g., $(A \to F \to B)_{f_1}$ depends on $(A \to B)_{f_2}$ while $(C \to G \to D)_{f_2}$ depends on $(C \to D)_{f_1}$.

To resolve local dependencies, we would like to find the exact local critical nodes controlling traffic on potential congested links for each flow. Thus, the updates of these nodes are critical to avoid congestions locally.

Lemma 2. *For flows $\forall f \in F_n(l) \cup F_o(l)$ on a potential congested link $l = u \to v \in P_n(f) - P_o(f)$ or $P_o(f) - P_n(f)$, there must exist at least a critical node $\exists n_f \in C(f)$ preceding u on $P_n(f)$ or $P_o(f)$ respectively.*

Proof. If $u \in C(f)$, $n_f = u$. Otherwise, $u \notin C(f)$:

For $\forall f \in F_o(l)$, $l \notin P_n(f)$. If we can not find a critical node from $P_o(f)[0]$ to u along $P_o(f)$, as the source node $P_o(f)[0] \in P_n(f) \cap P_o(f)$ and $P_o(f)[0] \notin C(f)$,

$rule(P_o(f)[0], P_n(f)) = rule(P_o(f)[0], P_o(f))$, thus link $P_o(f)[0] \to P_o(f)[1] \in P_n(f)$. Iteratively, $P_o(f)[1] \to P_o(f)[2], ..., u \to v \in P_n(f)$, which conflicts with $l \notin P_n(f)$. Therefore, there must $\exists n_f \in C(f)$ between $P_o(f)[0]$ and u.

For $\forall f \in F_n(l)$, $l \notin P_o(f)$. According to Algorithm 1, any switch in the new path belongs to a segment sg, and the beginning node of the segment $sg.start \in C(f) \cup \{P_n(f).start\}$. If $sg.start \in C(f)$, $n_f = sg.start$. Otherwise, $sg.start = P_n(f).start \notin C(f)$. If we can not find a critical node from $P_n(f)[0]$ to u along $P_n(f)$, similar with the proof of flows in $F_o(l)$, $u \to v \in P_o(f)$, which conflicts with $l \notin P_o(f)$. Thus, there must $\exists n_f \in C(f)$ between $P_n(f)[0]$ and u. $\square$

As we always could find a preceding critical node for each flow on a potential congested link with Lemma 2, these critical nodes control the amount of traffic of each flow on the potential congested link. Especially during the multipath transition [22, 17, 38], the critical node splits traffic of a flow between the old subpath and new subpath and shifts traffic gradually to the new path with different weights on the new path and old path. Therefore, the dependency among segments could be transformed into dependency among critical nodes controlling traffic on each potential congested link.

Definition 2 (Critical nodes dependency for potential congested link $l = u \to v$)**.** *The last critical node $n_f(l)$ of each flow $f \in F_n(l) \cup F_o(l)$ preceding u controls traffic on l. Thus, the critical node set of $F_n(l)$ depends on the critical node set of $F_o(l)$: $CN(F_n(l)) = \{n_{f_n}(l) | \forall f_n \in F_n(l)\} \rightharpoonup CN(F_o(l)) = \{n_{f_o}(l) | \forall f_o \in F_o(l)\}$.*

We use $\rightharpoonup$ to indicate the dependency in Definition 2. With local critical nodes dependencies for potential congested links, we get a dependency graph in which each potential congested link l matches a directed edge from the critical node set

Table 2.2: Dependency graph

Figure	Dependency Graph
2.1b	$A_{f_2} \rightharpoonup A_{f_1} \quad C_{f_1} \rightharpoonup C_{f_2}$
2.1c	$A_{f_1} \rightleftharpoons A_{f_2} \leftharpoonup E_{f_1} \quad C_{f_1} \rightharpoonup C_{f_2}$
2.1d	$A_{f_1} \rightleftharpoons A_{f_2} \quad C_{f_1} \rightleftharpoons C_{f_2} \quad E_{f_1} \rightleftharpoons E_{f_2}$

$CN(F_n(l))$ to $CN(F_o(l))$. The critical nodes dependency of link $A \rightarrow B$ in Figure 2.1b is $A_{f_2} \rightharpoonup A_{f_1}$ in which A_{f_2} and A_{f_1} are critical nodes of f_2 and f_1 respectively for link $A \rightarrow B$. Likewise, the dependency graphs for the three updating scenarios in Figure 2.1 are showed in Table 2.2. For the dependency graph of Figure 2.1b, A_{f_1} and C_{f_2} should update before A_{f_2} and C_{f_1} to resolve the two dependencies independently. With independent segments in Table 2.1, the updates in Figure 2.1b could be scheduled by updating segments $(A \rightarrow F \rightarrow B)_{f_1}$, $(C \rightarrow G \rightarrow D)_{f_2}$ before $(A \rightarrow B)_{f_2}$ and $(C \rightarrow D)_{f_1}$, so that A_{f_1} and C_{f_2} update before A_{f_2} and C_{f_1}.

2.5 Dependency Resolution

While the global dependency among updates is divided into local dependencies among critical nodes to reduce resolution complexity, switches in each segment should also follow the reverse order updating to preserve black-hole free and loop free consistency. In this section, we design a heuristic algorithm to resolve the dependency graph while considering update order inside each segment. The key notations of the following algorithms are summarized in Table 2.3.

2.5.1 Direct Dependency Resolution

After constructing the dependency graph (Line 5-8) in Algorithm 2, the nodes which do not belong to the dependency graph could be updated with $UpdateSegment(sg)$

Table 2.3: Key notations in Cupid

Notation	Definition
f	flow
$b(f)$	bandwidth requirement of flow f
$c(l)$	bandwidth capacity of link l
$a \rightarrow b$	a precedes b along a forwarding path
$a \rightharpoonup b$	a depends on b
$P_o(f)$	the old forwarding path of flow f
$P_n(f)$	the new forwarding path of flow f
$C(f)$	critical nodes $C(f)$ of a flow f
$Sg(f)$	segments of a flow f
$F_o(l)$	flows to be moved away from link l
$F_n(l)$	flows desire to use link l
$F_u(l)$	unchanged flows going through link l
n_f	node n on the forwarding path of flow f
$d(l)$	dependency among critical nodes for potential congested link l
$CN(F_n(l))$ or $d(l).CN(F_n)$	critical node set of $F_n(l)$
$CN(F_o(l))$ or $d(l).CN(F_o)$	critical node set of $F_o(l)$
$n_f.old/n_f.new$	the amount of traffic on the old/new path controlled by node n for flow f

(Line 9) and added to the update sequence US, as long as the downstream nodes in the same segment have already been updated.

For the nodes involved in the dependency graph D, if the update of a critical node $n_f \in d(l).CN(F_o)$ does not depend on others, which means no other $d(l').CN(F_n)$ contains n_f and the downstream nodes in the same segment have been updated ($CanUpdateInSegment(n_f)$), n_f could be updated immediately (Line 11-15). After the update of n_f, nodes in the same segment sg previously blocked by n_f due to the reverse order updating are able to update with $UpdateSegment(sg)$, and then n_f is removed from the dependency graph (Line 14). Furthermore, if the updates of some nodes in $d(l).CN(F_o)$ relieve potential congestions on l (Line 16),

Algorithm 2 Direct Dependency Resolution

```
1: # Initialization:
2: $US = \varnothing$ #update sequence
3: $D = \varnothing$ #dependency of critical nodes
4: $CL$ = potential congested links
5: for each link $l : CL$ do
6:     $d(l) = CN(F_n(l)) \rightharpoonup CN(F_o(l))$
7:     $D = D \cup d(l)$
8: end for
9: $US = US + UpdateSegment(sg)\ (\forall f, sg \in Sg(f))$
10: # Critical node $n_f$ update without deadlocks:
11: if $(\exists d(l) \in D) \wedge (d(l).CN(F_o) = \varnothing$ or $n_f \in d(l).CN(F_o) \wedge n_f \notin \forall d(l').CN(F_n))$
    then
12:     if $CanUpdateInSegment(n_f)$ then
13:         $US = US + n_f + UpdateSegment(sg)\ (n_f \in sg)$
14:         remove $n_f$ from all $D$
15:     end if
16:     if $IsPotentialCongested(l) == false$ then
17:         for $n_f : d(l).CN(F_n) \cup d(l).CN(F_o)$ do
18:             if $CanUpdateInSegment(n_f) \wedge n_f \notin \forall d(l').CN(F_n)$ then
19:                 $US = US + n_f + UpdateSegment(sg)\ (n_f \in sg)$
20:                 remove $n_f$ from $D$
21:             end if
22:         end for
23:         $D = D - d(l),\ CL = CL - l$
24:     end if
25: end if
26: # Schedulable critical node $n_f$ update in deadlock:
27: if $(\exists d(l) \in D) \wedge (n_f \in d(l).C(F_o) \wedge InDeadlock(n_f) \wedge CanSchedule(n_f))$ then
28:     if $CanUpdateInSegment(n_f)$ then
29:         $US = US + n_f + UpdateSegment(sg)\ (n_f \in sg)$
30:         remove $n_f$ from $D$
31:     end if
32: end if
```

Line 17-22 update free critical nodes in $d(l).CN(F_n)$ and $d(l).CN(F_o)$, and Line 23 removes the dependency $d(l)$ from dependency graph D and deletes l from potential congested link set CL.

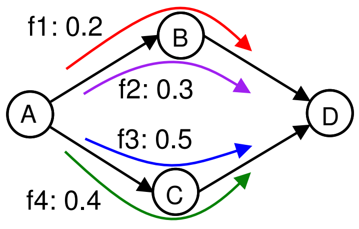
Although the dependency $C_{f_1} \rightharpoonup C_{f_2}$ in Figure 2.1c could be scheduled sequentially with Line 11-25 in Algorithm 2, A_{f_1} and A_{f_2} depend on each other which makes a deadlock. The deadlocks in Table 2.2 are clear, as each critical node set has only one node. However, the critical node sets of each link may contain several critical nodes. We define the deadlock as a cycle in which $CN(F_o)$ and $CN(F_n)$ of adjacent critical node dependencies share at least one critical node.

Definition 3 (Deadlock L)**.** *For a set of potential congested links* $\{l_0, l_1, l_2, ..., l_k\}$, $CN(F_o(l_0)) \cap CN(F_n(l_1)) = cn_0 \neq \varnothing$, $CN(F_o(l_1)) \cap CN(F_n(l_2)) = cn_1 \neq \varnothing$, ..., $CN(F_o(l_k)) \cap CN(F_n(l_0)) = cn_k \neq \varnothing$, *the intersection critical node sets* $cn_0, cn_1, ..., cn_k$ *form a deadlock* L*.*

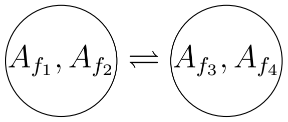
Actually, not all the nodes involved in a deadlock are blocked. As Figure 2.2 shows, there are 4 flows from A to D using $A \to B \to D$ and $A \to C \to D$ respectively. If we want to exchange the routing paths of f_1, f_2 with f_3, f_4 to release more available bandwidth for other flows on link $A \to C$, the exchanging is recognized as a deadlock in Figure 2.2b. Indeed, the exchanging could be serialized with the update sequence f_3, f_1, f_2, f_4, as the available bandwidth on new paths is large enough to reroute flows directly. Therefore, schedulable nodes in a deadlock may be updated directly with an appropriate update order. In Algorithm 2, Line 27 checks whether nodes in deadlocks ($InDeadlock(n_f)$) could be scheduled with enough bandwidth ($CanSchedule(n_f)$), and then add schedulable nodes to the update sequence in Line 28-31.

2.5.2 Updating With Multipath Transition

Although schedulable critical nodes could update directly to resolve the deadlock in Figure 2.2, there may be situations in which no node in a deadlock could update completely in a step due to the limitation of bandwidth capacity. For example, if



(a) Current state



(b) Dependency graph

Figure 2.2: Schedulable flows in deadlock: the link bandwidth capacity is 1 unit, and $f : b$ means the throughput b of flow f.

throughputs of f_2 and f_3 are both 0.4 unit, none of these 4 flows could be updated directly. Therefore, we use multipath transition by spitting traffic of a flow between the old path and new path with critical nodes. For the deadlock $A_{f_1} \rightleftharpoons A_{f_2}$ in Figure 2.1c, we split both flows f_1 and f_2 with link $A \to B$ and $A \to E$ to shift traffic to their new paths gradually. To reduce the multipath transition overhead, we design a greedy algorithm to shift the largest amount of traffic to minimize multipath transition steps. For example, A_{f_1} in Figure 2.1c shifts 0.5 unit of f_1 to $A \to E$ firstly, as 0.5 unit on $A \to E$ available for f_1 is larger than 0.2 unit for f_2 on $A \to B$. In Algorithm 3, Line 2 searches the non-blocked critical node with the largest available bandwidth ab in deadlock L. The available shifting bandwidth ab for n_f is the minimum value of the amount of traffic on the old path $n_f.old$ and minimum available bandwidth along the new subpath. If $ab > 0$, Line 6-7 reassign weights $n_f.old : n_f.new$ for the old path and new path, and add the node to update sequence. With the multipath migration, all the traffic on the old path is finally shifted to the new path (Line 8), so that Line 10 removes n_f from the dependency graph. After the completion of multipath transition in Figure 2.1c, E_{f_1} is able to update as the dependency on A_{f_2} has been resolved.

Algorithm 3 Multipath Transition

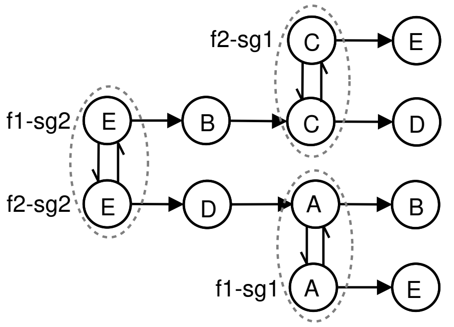
```
1: while $L \neq \varnothing$ do
2:     $\{n_f, ab\} = \max_{n_f \in L}\{\min\{n_f.old, \min_{n_f \in \forall d(l').CN(F_n)} AvailBw(l')\}\} \wedge CanUpdateInSegment(n_f)$
3:     if $ab \leq 0$ then
4:         $RateLimit(f_r = \max_{f' \in P_n(f)}(b(f')))$
5:     end if
6:     $n_f.old : n_f.new = (n_f.old - ab) : (n_f.new + ab)$
7:     $US = US + n_f$
8:     if $n_f.old == 0$ then
9:         $US = US + UpdateSegment(sg)$ $(n_f \in sg)$
10:        $L = L - n_f$, remove $n_f$ from $D$
11:    end if
12: end while
```

Unfortunately, multipath transition may be blocked by fully utilized links. For example, in Figure 2.2, if throughputs of all the 4 flows are 0.5 unit, no flow could migrate as there is no available bandwidth at all. In this case, the maximum available bandwidth $ab \leq 0$ (Line 3), so that we need to limit throughputs of some flows to release a small amount of available bandwidth for multipath transition. Thus, if we would like to shift f_1 with multipath in Figure 2.2, we should limit rate of flows f_3 or f_4 on the new path of f_1, e.g., reducing 0.2 unit of f_3, so that f_1 could split traffic with weights 0.2 : 0.3 on the new path and old path, and then multipath transition is able to carry out as normal. After the multipath transition, the throughputs of rate-limited flows are restored to reduce throughput losses.

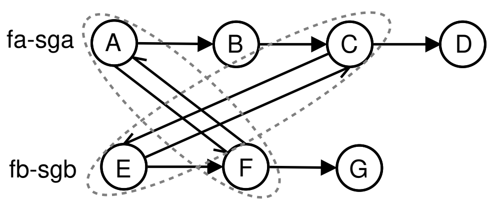
2.5.3 Dependency Resolution With Segments

2.5.3.1 Feasibility of Dependency Resolution

For Figure 2.1d, the dependency graph requires multipath transitions for (A_{f_1}, A_{f_2}), (C_{f_1}, C_{f_2}) and (E_{f_1}, E_{f_2}) as Table 2.2 indicates. While a multipath transition updates



(a) Dependencies and segments in Figure 2.1d



(b) Conflicts between dependency graph and segments

Figure 2.3: Combining dependency graph with segments: f-sg means segment sg of flow f

multiple nodes in a deadlock simultaneously, nodes should follow reserve order updating in each segment, which is checked by $CanUpdateInSegment(n_f)$ in Algorithm 3. Combining the dependency graph and segments of Figure 2.1d together, Figure 2.3a shows that multipath transitions of (A_{f_1}, A_{f_2}) and (C_{f_1}, C_{f_2}) are independent from each other, so that these two deadlocks could be scheduled concurrently as long as their downstream nodes have been updated. After the resolution of $A_{f_1} \rightleftharpoons A_{f_2}$ and $C_{f_1} \rightleftharpoons C_{f_2}$, switch D in segment sg_2 of f_2 and switch B in segment sg_2 of f_1 are able to update, and then the multipath transition of (E_{f_1}, E_{f_2}) resolves the dependency graph finally.

Even though dependency graph could be resolved following reverse order updating of each segment in Figure 2.3a, there are still situations in which dependency can not be solved due to conflicts between the dependency graph and segments. Figure 2.3b requires multipath transitions for (A_{f_a}, F_{f_b}) and (C_{f_a}, E_{f_b}). Meanwhile, the reserve order updating of segments sg_a and sg_b instructs that C_{f_a} updates before A_{f_a} and F_{f_b} updates before E_{f_b} respectively. Consequently, the multipath transition of (A_{f_a}, F_{f_b}) has to wait for the update completion of C_{f_a}, while the update of C_{f_a} in multipath transition of (C_{f_a}, E_{f_b}) requests F_{f_b} in (A_{f_a}, F_{f_b}) to update firstly. Thus, there is a conflict between dependency graph and segment. We detect this kind of conflicts by checking cycles formed by the dependency graph and reserve order of segments. The cycle $A \rightarrow B \rightarrow C \rightleftharpoons E \rightarrow F \rightleftharpoons A$ in Figure 2.3b is composed of critical nodes dependencies and sub-segments, while there is no cycle formed in Figure 2.3a.

Theorem 3. *If no cycle forms with the dependency graph and reverse order of segments, we could always find an update order with Algorithms 2 and 3.*

The update scheduling for the dependency graph and reverse order of segments forming no cycle is as follows: Algorithm 2 first updates free nodes and schedulable nodes in deadlocks, and then Algorithm 3 resolves deadlocks with multipath transition and rate-limit. After resolving deadlocks, nodes previously blocked by deadlocks are now relieved and could be scheduled by running Algorithm 2 again. Iteratively, all the updates and deadlocks are scheduled with Algorithm 2 and 3.

2.5.3.2 Dependency Resolution And Update Efficiency

The dependency resolution generates an update order for rules. As different update orders would result in different update delays in the data plane, scheduling a fast update order is critical to reduce the duration of updating. Intuitively, the update delay depends on the updates in the most complicated dependency chain. In Figure 2.3a, the update time depend on segments $E \rightarrow B \rightarrow C \rightarrow D$ of f_1, $E \rightarrow D \rightarrow A \rightarrow B$ of f_2, and critical nodes dependency $E_{f_1} \rightleftharpoons E_{f_2}$. Thus, D_{f_1} and B_{f_2} should start to update as early as possible, so that the finishing time could be earlier. However, the complexity of dependency chain is difficult to estimate. We design a heuristic scheduling algorithm (Algorithm 4) for dependency resolution based on the length of the dependency chain. The dependency chains consist of not

Algorithm 4 Efficient Dependency Resolution Scheduling

```
1: $DGraph$ = dependency graph $\cup$ segments
2: for each pair $d(l_1), d(l_2) \in D$ do
3:     if $d(l_1).CN(F_n) \cap d(l_2).CN(F_o)$ then
4:         $DGraph = DGraph \cup (d(l_1) \rightarrow d(l_2))$
5:     end if
6: end for
7: while $DGraph.nodes \neq \varnothing$ do
8:     $Nodes$ = sort $node \in DGraph.nodes$ in decreasing order of $dclength(n)$
9:     for $node \in Nodes$ do
10:        resolve $node$ with Algorithm 2 and 3
11:    end for
12: end while
```

only the dependency graph and segments but also the dependencies among critical nodes dependencies (Line 1-6). We select the nodes with long dependency chains for scheduling first (Line 7-12), and expect that the nodes at the beginning of these long dependency chains could be updated as early as possible with less short update completion delays. The *dclength* of *node* is defined as the length of upstream dependency chain depending on *node* in (2.4), which is the sum of the weights of nodes on the upstream chain. For a critical nodes dependency *node*, the weight is the number of nodes in $CN(F_o)$ and $CN(F_n)$. Otherwise, for a single update node, the weight is 1.

$$weight(node') = \begin{cases} size(node'.CN(F_o)) + size(node'.CN(F_n)) & node' \in D \\ 1 & node' \notin D \end{cases} \tag{2.3}$$

$$dclength(node) = \max_{chain.end=node} \sum_{node' \in chain} weight(node') \tag{2.4}$$

Although the sequential update in the data plane could ensure the correctness, it hinders the update efficiency. As rules on different switches could update concurrently, we note that parallel update of rules will reduce the overall update delay. When we resolve a rule from dependency, we also assign it with the exact update time slot on the concerned switch to ensure the correct update order in the data plane. Assuming a switch could only update one rule at a time slot. The update time of rules not only depends on the dependency resolution sequence but also the available slots in the related switches. We should update n_f after the update of the next-hop of n_f in the same segment ($slot_{n_f \to m_f \in sg \in Sg(f)}(m)$) and the currently maximum update time $d(l).CN(F_n).maxslot$ of scheduled old rules on which n_f depends ($n_f \in d(l).CN(F_n)$). Moreover, switch $n_f.sw$ should update n_f with an empty time slot ($n_f.sw.atslot(t) = null$). We update n_f at the first time ($slot(n_f)$) when these conditions are satisfied to ensure updates are applied as soon as possible with (2.5).

$$
\begin{aligned}
slot(n_f) = \min\{t | t > \max\{&slot_{n_f \to m_f \in sg \in Sg(f)}(m), \max_{\forall l, n_f \in d(l).CN(F_n)} d(l).CN(F_n).maxslot\}, \\
&n_f.sw.atslot(t) = null\}
\end{aligned}
\tag{2.5}
$$

2.6 Implementation

We implement Cupid with 2000+ lines of Java code. Cupid sits between the routing modules (e.g., failure recovery, load balancing) and the control message communication module in the controller. In the architecture showed in Figure 2.4, all control messages to manipulate forwarding rules in flow tables are captured to schedule an appropriate updating order before applied in switches. The new routing path

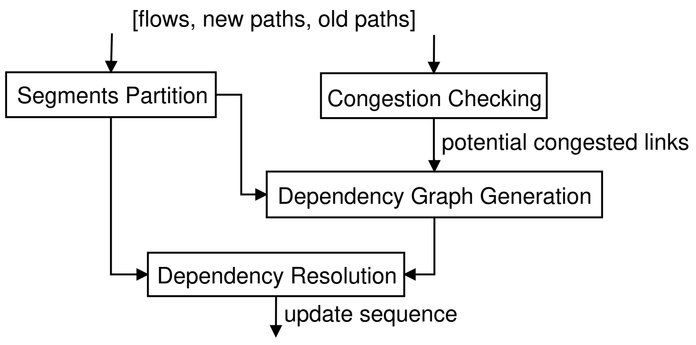

[flows, new paths, old paths]
Segments Partition
Congestion Checking
potential congested links
Dependency Graph Generation
Dependency Resolution
update sequence

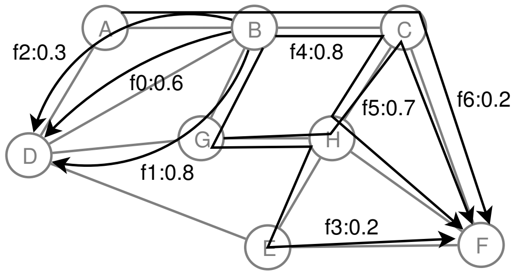
Figure 2.4: Cupid architecture

of each flow is partitioned into several independent segments with Algorithm 1. We identify potential congested links with Definition 1, and then generate a dependency graph among critical nodes for these links with Definition 2. Loops in routing paths and deadlocks in the dependency graph are recognized with strong connected components. Finally, with Algorithm 2, 3 and 4, a feasible and efficient updating order is scheduled to update flow tables consistently without congestions.

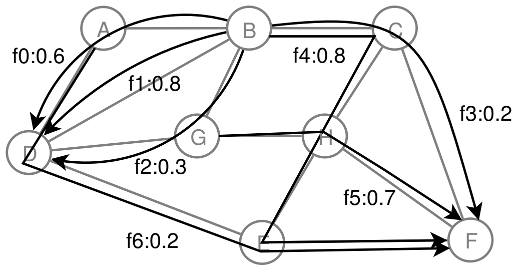
Instead of two-phase commit update, we use only one rule for a flow at any time with multiple ports in input and outport fields. The outport field of flow entries allow multiple output ports with different weights on critical switches. During the migration, the critical nodes shift traffic to the new path with assigned weights of output ports on the old path and new path respectively. As a downstream switch may have been updated while packets arrive at the old input port, the inport field also allows multiple input ports to make sure all packets arrived are handled by switches correctly. We set timeout for ports of old path, so that the old ports are deleted when expire in a flow entry.

2.7 Case Study

We first provide a case study to show the advantages of Cupid. We study a network in Figure 2.5 with 8 switches shifting the forwarding paths of 7 flows from



(a) Initial state



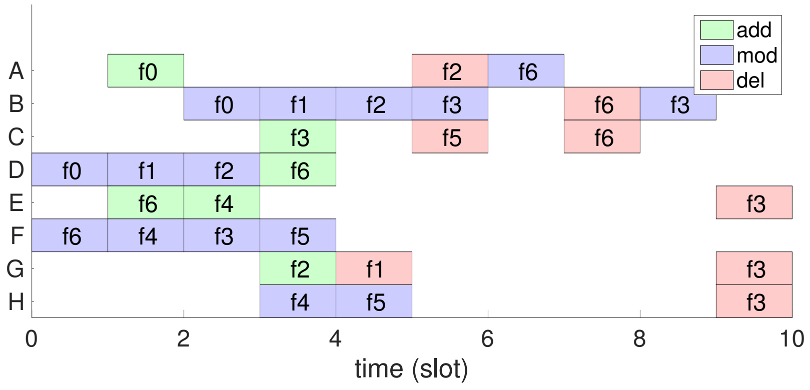
(b) Target state

Figure 2.5: Case study

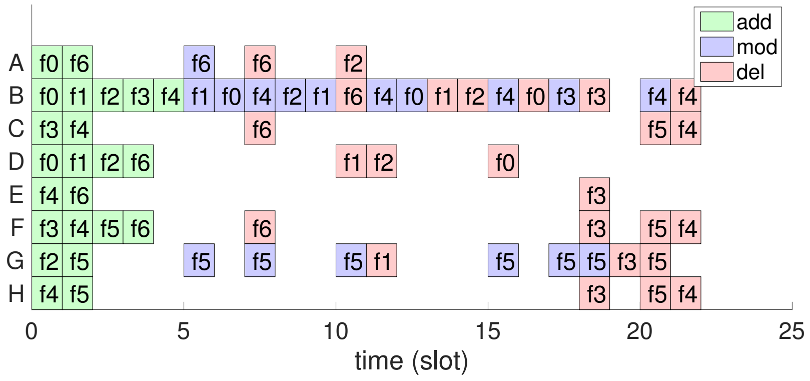
initial state to target state. The throughputs of flows are shown on the figure, and the bandwidth capacities of links are 1 unit. We compare the performance of Cupid with Dionysus [22] in the flows migration.

Update Delay In Data Plane: Figure 2.6 shows the timeline of applying updates in switches for Cupid and Dionysus. Although these two update orders both ensure the correctness of updating, the update time of Cupid is only 10 slots, which is much shorter than 22 slots of Dionysus. This is due to two main reasons: 1) we divide the global dependency into local dependencies so that a lot of independent rules in different segments could update in parallel; 2) as we keep multiple inports and outports in each rule entry, there is only one flow entry for each flow in a switch at any time, and most actions in Cupid are modifications to inport and outport fields, while Dionysus has to add a new rule for a flow in each related switch and then modify the weights of the new rule and old rule to migrate traffic before finally deleting the old rule, which leads to more operations in the data plane.

Flow Table Space In Switches: Because of the single flow entry with multiple inports and outports, Cupid always consumes a less number of flow table entries than Dionysus during updating in Figure 2.7, so that Cupid promises to save flow table space in data plane, which is encouraging for the limited flow table space.



(a) Cupid



(b) Dionysus

Figure 2.6: Update timeline

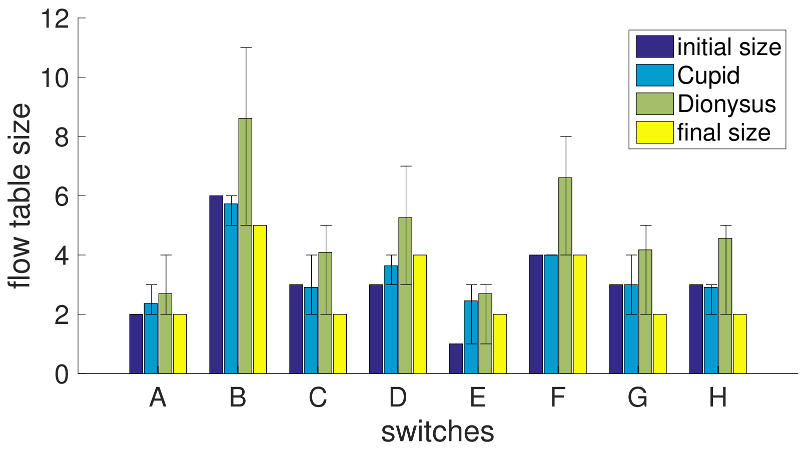


Figure 2.7: Average flow table size during updating

2.8 Evaluation

2.8.1 Evaluation Setup

We evaluate Cupid in fat-tree and mesh networks with simulation. These two topologies have different path deployments and traffic distributions, so that the dependencies imposed by congestion-free consistency are quite different. Each network consists of 100 switches connected by 1Gbps links.

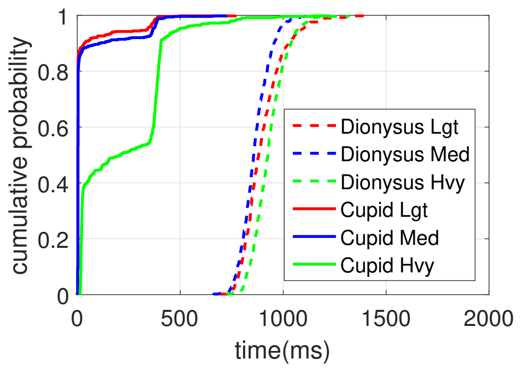
- Fat-tree: We use a three-layer fat-tree topology [48] with 66 edge switches, 22 aggregate switches and 12 core switches, so that link bandwidth capacities of aggregated layer and core layer are balanced. Traffic distribution in the fat-tree network follows the data center traffic characteristics in [49].
- Mesh: The diameter of the mesh topology is 18, and the degree of each node in the network is 4. The traffic in the mesh network is randomly generated and uniformly distributed among all node pairs.

More than 10,000 flows are running simultaneously in each network. During simulation, we assign link failures in the network and reroute affected flows to other available paths, and also schedule flows to other less loaded paths for load balancing. The simulations are evaluated with quad-core 2.4GHz processor and 8GB RAM. We compare the update ordering efficiency of Cupid with Dionysus [22] and random update ordering.

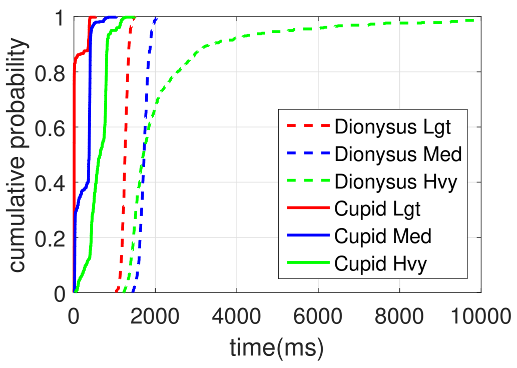
2.8.2 Evaluation Results

2.8.2.1 Update Ordering Latency

Figure 2.8 shows the ordering latency of updating 1000 flows simultaneously under light (<30% network utilization), medium (30∼70% network utilization) and heavy (>70% network utilization) traffic load. The network utilization is measured by weighted link utilization. In both fat-tree and mesh networks, Cupid takes shorter



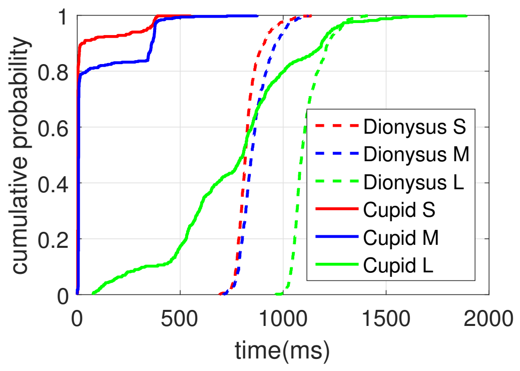
(a) Fattree



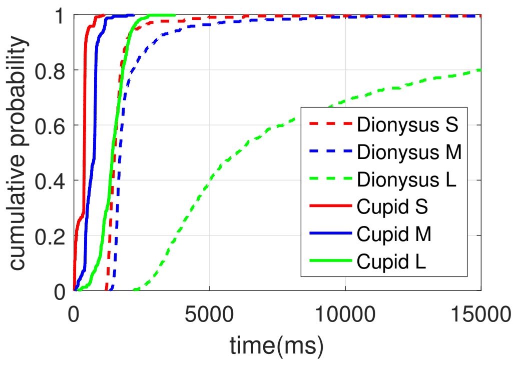
(b) Mesh

Figure 2.8: Ordering latency under different traffic load

time to schedule a feasible updating order than Dionysus. With divided independent segments and local dependencies among critical nodes, Cupid reduces dependency resolution complexity and ensures a faster update ordering. For the three layer fat-tree network, the longest routing path is only 4 hops and each switch has its own dedicated links to the higher and lower layer switches, so that flows have fewer chances to collide to congest links during updating. Thus, the dependency graph is relatively simple due to the short and dedicated routing paths in the fat-tree. Therefore, Cupid could finish ordering within 500ms in most cases while Dionysus solves the global dependency graph in 1000ms. On the other hand, the mesh network tends to encounter more congestions during updating compared with fat-tree, especially on links in the middle of network shared by a lot of flows. Meanwhile, the routing paths in the mesh topology are usually longer than 4 hops in the fat-tree, and there may be loops formed with new and old paths during updating, which adds to the complexity of dependency graph. For the light and medium traffic load, although Dionysus could schedule the ordering within 2000ms, the ordering time of Cupid is 500ms which is 4 times faster than Dionysus. The situation is much worse for



(a) Fattree



(b) Mesh

Figure 2.9: Ordering latency with different flow size

Dionysus under heavy traffic load. As the highly complex dependency graph is more difficult to resolve due to the scarcity of available bandwidth, Dionysus takes even tens of seconds to find a feasible ordering, while Cupid is still able to schedule the ordering within 1000ms.

Although a lot of researches [32, 33, 34] discover and reroute large flows to less congested path for load balancing, large flows are more likely to be stuck by limited bandwidth resource during updating. Thus, large flows tend to migrate to new paths with multipath transition, while small flows probably could be scheduled freely with a small amount of available bandwidth. To show this difference in updating, we classify flows into three classes according to flow size: small flow (<1M), medium flow (1∼10M) and large flow (>10M). Figure 2.9 shows the order scheduling time for 1000 flows with the three classes under heavy traffic load. In the fat-tree topology, the update ordering of small and medium flows in Cupid takes much shorter than Dionysus. However, the ordering of large flows updates in Cupid may be worse than Dionysus at times. Figure 2.10 shows most of flows in the fat-tree topology have only one segment, as the ingress and egress switches are the only common nodes in

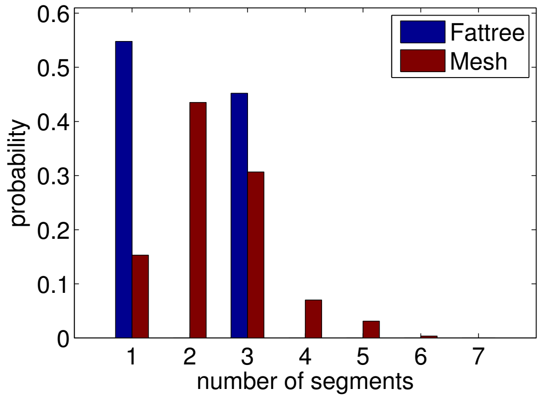


Figure 2.10: Segments count

the new and old paths, so that the segment partition brings few benefits for fat-tree. Moreover, with large flows under heavy traffic load, Cupid identifies almost all the links as potential congested links and constructs dependencies for these links, so that the resolution of the large dependency graph takes longer. Nevertheless, the overall update ordering time of Cupid is much shorter than Dionysus in most cases (over 90%). Compared with the fat-tree topology, flows usually have longer routing paths in mesh network and also larger number of segments as Figure 2.10 shows. With the benefits of independent segments and local dependencies, Cupid always outperforms Dionysus in mesh network. Especially for large flows, Dionysus takes even tens of seconds to resolve the complicated global dependency graph, while Cupid is able to finish ordering within 2000ms.

2.8.2.2 Dependency Resolution Analysis

To understand how dependencies are resolved, we divide the resolution process into 4 phases: non-deadlock, sched-deadlock, multipath and rate-limit, which correspond to the updates without deadlock, schedulable updates in deadlocks, multipath transition and rate-limit respectively in Algorithm 2 and 3. Figure 2.11 shows phases

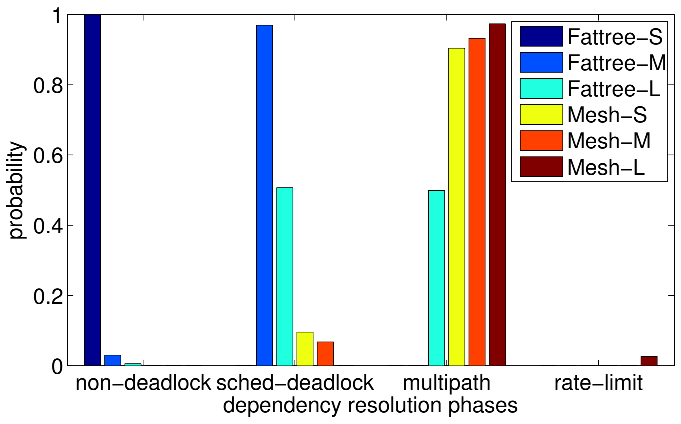
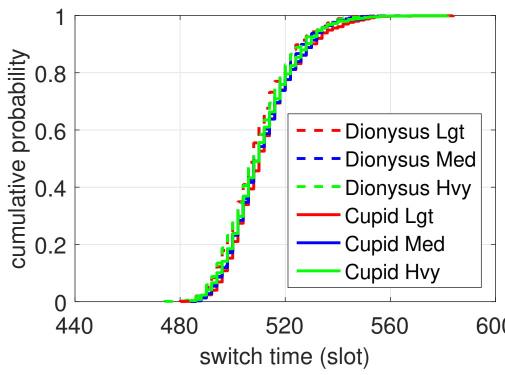


Figure 2.11: Dependency resolution phases

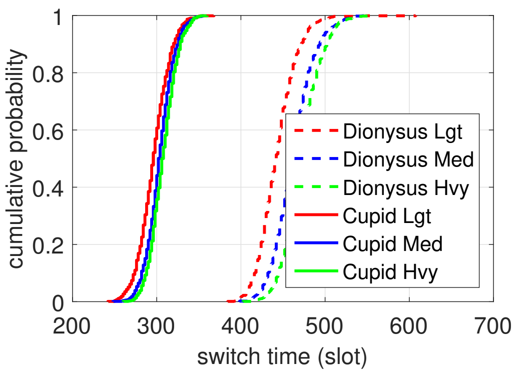
at which dependencies are resolved for different size of flows under heavy traffic load. As small flows almost could be freely scheduled in the fat-tree network, there are few potential congestions and few flows are involved in dependency resolution, so that these dependencies could be solved in the non-deadlock phase. Compared with the mesh topology, the update ordering in fat-tree could be scheduled in the first three phases, while updates in mesh network tend to be scheduled in the last three phases due to the higher dependency complexity. Especially for the large flows updating in the mesh network, over 2% orderings fall into the rate-limit phase which results in throughput losses.

2.8.2.3 Data Plane Update Delay

The update ordering calculates an update sequence for the rules in the data plane. Figure 2.12 and 2.13 show the update delay in the data plane when applying the update sequences in switches under different traffic load and with different flow size respectively. As we schedule the update ordering based on the length of dependency chain in Cupid, updates with long downstream dependency chains are scheduled as early as possible to reduce the completion time. Updates in Cupid

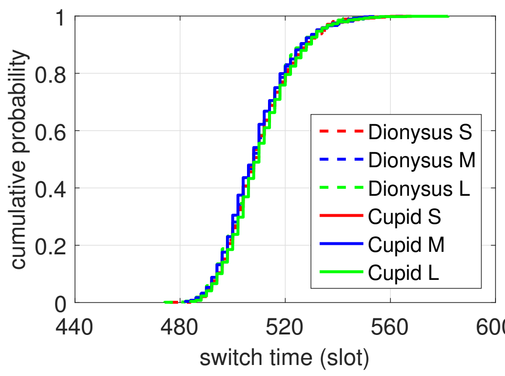


(a) Fattree

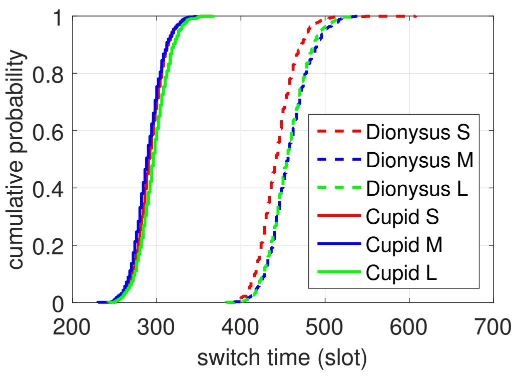


(b) Mesh

Figure 2.12: Switch update latency under different traffic load



(a) Fattree



(b) Mesh

Figure 2.13: Switch update latency with different flow size

always take less time to take effect than Dionysus in mesh network. The divided local dependency contributes to the parallelism of updating. Updates are scheduled as soon as the concerned switches have available slots after the dependency on other rules are resolved. These ensure a highly parallel and efficient updating in the data plane. However, for the fat-tree network, the data plane update latency of Cupid is similar to Dionysus. As most flows in fat-tree have a single segment, this requires sequential reverse order update of the new paths. Moreover, flows usually go through shared core switches and aggregate switches in fat-tree, and updates in

Table 2.4: Network utilization losses (90th percentile)

Topology	Approach	Small	Medium	Large
Fattree	Cupid	0	0	0
	Dionysus	0	1.59×10^{-16}	2.46×10^{-9}
	Random	0	0.06%	0.48%
Mesh	Cupid	0	0	0.16%
	Dionysus	0.12%	0.79%	4.71%
	Random	0.54%	1.41%	7.39%

these switches are restricted by available time slots. Therefore, although the update ordering latency takes shorter time due to the less complicated dependency in fat-tree, updates take longer to be applied in the data plane than the mesh network. Unlike the update ordering latency, the data plane update delay does not vary a lot when the traffic load or flow size changes. Especially for Cupid, as we update rules as early as possible with available time slots, the data plane update process is highly compacted and the switch time slots are efficiently utilized.

2.8.2.4 Throughput Losses

Due to the limited link bandwidth, flows have to reduce their throughputs once fall into the rate-limit phase during updating. Cupid could schedule updates without any throughput loss in non-deadlock, sched-deadlock and multipath phases. Only a small percentage of orderings fall in the rate-limit phase as Figure 2.11 shows. We compare the network utilization losses of Cupid with Dionysus and random update ordering. In Table 2.4, there are always less network utilization losses with Cupid than Dionysus and random update ordering in both fat-tree and mesh networks. Even though the losses in the fat-tree network are quite low using Dionysus, Cupid does not experience any loss. Moreover, for the large flows migration in the mesh

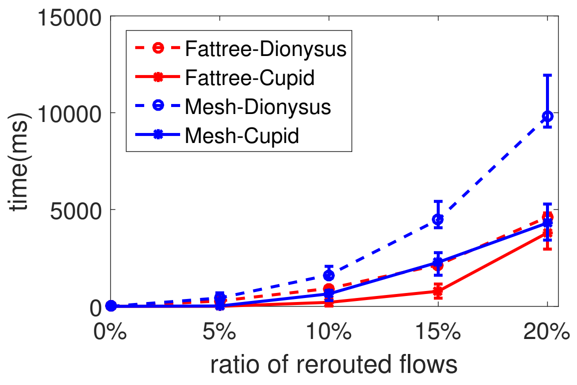


Figure 2.14: Ordering latency with different number of flows (50th percentile with [10th, 90th])

network, the network utilization loss in Cupid is 0.16% while the losses of Dionysus and random update order are 10 times larger than Cupid.

2.8.2.5 Scalability

We further study how the update ordering latency scales with the number of involved flows under heavy traffic load, and find that Cupid always schedules faster than Dionysus when migrating 0∼20% of flows in the network to new paths in Figure 2.14. Especially for mesh topology, the ordering time in Dionysus rises steeply with the increasing number of involved flows, as more flows adds to the global dependency graph size and complexity. Cupid is able to schedule the ordering within 4000ms, while Dionysus takes more than 10 seconds for updating 20% flows in mesh network.

2.9 Conclusion

With increasing SDN applications scheduling flows for load balancing and failure recovery, in this chapter, we focus on updating flow tables in data plane consistently and efficiently while preserving throughputs of flows. To reduce the overhead of finding a feasible updating order, we firstly partition the rerouted path into independent segments, and then divide the global dependency among updates into local dependencies among critical nodes. We then design and implement a heuristic dependency resolution algorithm with the dependency graph and reverse order updating within each segment. To reduce the flow table space overhead, we use multiple ports with weights in each flow entry during updating, so that there is only one rule kept for each flow in a switch. The results of simulation show that Cupid is able to schedule update ordering at least 2 times faster than Dionysus and has less throughput losses in both fat-tree and mesh topologies.

CHAPTER 3
Redactor: Reconcile Network Control With Declarative Control Programs

This chapter shows how control programs generate consistent control decisions to ensure correct network behaviors. The controller is responsible for composing multiple control programs and policies into a single consistent policy before applying it in the data plane. Otherwise, contradictory control decisions made by control programs for different objectives would misconfigure the network. Moreover, large SDN networks may consist of multiple controllers in different control domains, which necessitates the coordination of control decisions. Among the existing solutions, composing control programs dynamically is tricky, and coordinating decisions of independent control programs usually results in suboptimal decisions. In this chapter, we propose an approach for control reconciliation with declarative language Prolog instead of defining a new programming language. It could compose control programs easily with the declarative property and reconcile control conflicts with the knowledge base.

3.1 Introduction

Software-Defined Networking provides separated control logic from the forwarding plane with flexible control programs. These third-party control programs act as blackboxes to produce rules independently to control the network. However, the independence of control programs may lead to conflicts in their control decisions.

These conflicts must be resolved before being applied to data plane, otherwise, they may result in performance degradation or even unexpected network behaviors. Especially in large organizations, multiple policy sub-domains exist, e.g., server admins, network engineers, different departments, which set their own control programs to manage the network components [50]. With more number of entities generating control plans independently, the control conflicts would get more serious.

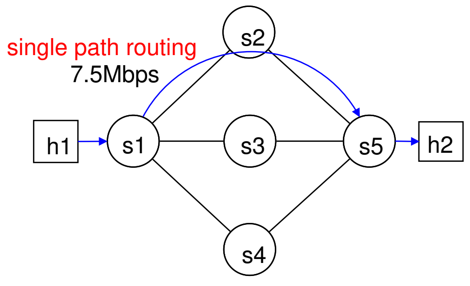
It is quite challenging to coordinate independent control programs unless negotiation process is allowed among control programs, which makes the implementation of control programs complicated and error-prone. Moreover, to avoid bottleneck at the control plane, SDN usually utilizes a logically centralized control plane with multiple distributed control nodes. It is more difficult to coordinate decisions made by distributed control nodes with remote negotiations due to the increasing number and diversity of control programs. Manual analysis of the interaction among control policies with different syntax and semantics is impractical [51]. Besides the control consistency, an efficient control decision should also satisfy as many control objectives as possible to ensure the network performance. As each control program aims at improving a sub-aspect of resource utilization, maximizing the control utility is expected to achieve the desired network performance to the most extent. Therefore, maintaining a consistent and efficient network control logic is critical to ensure the correctness and performance of the data plane.

To achieve control consistency, the controller has to reconcile requests and decisions of SDN control programs. As each control program generates solutions satisfying its own objective, the reconciliation of control programs is a kind of multiple objectives optimization. According to the point of optimization, the reconciliation approaches could be divided into two categories: beforehand control programs coordination which reconciles control intents before making decisions, and afterwards decisions checking which determines the rules to be applied after each control program generating its decisions.

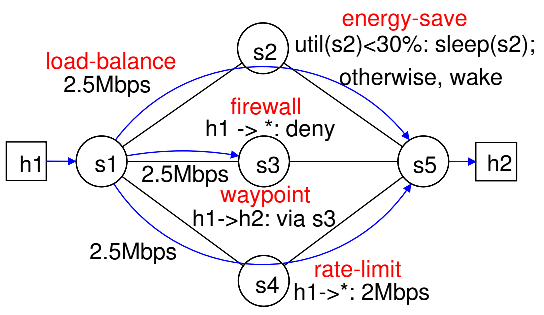
For the beforehand coordination, a lot of researches express intents of control programs at a high level of abstraction. They usually check relationships among intents [50] or dependencies on network resources [52, 53, 24]. As intents of control programs may change dynamically, e.g., a stateful firewall dropping flows after detecting the source host generating too many connections, the controller has to model and check the relationships of control intents in real time, which usually takes a long latency. These existing approaches devote a lot of efforts in maintaining consistency at the level of potential policies, instead of coordinating the behaviors of independent control programs to make a feasible control plan during the decision-making process. Despite the low efficiency of policies modelling and checking, an efficient way to reconcile control programs is composing diverse programs to produce a consistent policy. Although [54, 55, 56, 57] support module composition, they usually focus on certain types of programs, e.g., composing monitoring and routing programs, and require manual composition because of complicated implementation details. An automatic and dynamic composition approach of diverse control programs is absent. Moreover, these existing composition approaches fail to deal with control conflicts.

As network configurations are usually low-level, the afterwards decisions checking is widely used in control consistency maintenance, which usually resolves conflicts with priority or voting mechanisms [58, 59, 60]. With priority-based coordination, rules are arranged in order based on pre-defined prioritization which should be carefully designed. In voting approaches, control programs propose their own proposals independently which are probably suboptimal, and then the controller chooses the seemly "best" solution with the highest vote value but actually suboptimal among a set of suboptimal proposals. Moreover, neither prioritization nor voting ensures satisfying the maximum number of control objectives, as they both ignore the control utility in their control plan selection process. Verification is also used to afterwards examine the desired properties of control decisions [51, 61, 62], and it also could not ensure the best control utility.

Declarative languages provide an abstracted high-level way to express network-wide policies. While numerous authorization and verification approaches [63, 64, 65] have been proposed with declarative languages, few focuses on conflict checking and reconciling during the generating process of network rules. In this chapter, to reconcile network control automatically and dynamically, we propose the control program composition and coordination approach – Redactor, which makes consistent control decisions while maximizing the control utility. We implement SDN control programs with the declarative language Prolog, so that control programs could be composed with names and consistent requirements to execute together. To get better network performance, among the feasible decisions, we use the voting mechanism to decide the control plan with the most preferences of control programs. When conflicts occur, we design the control program compromise algorithms to resolve conflicts and maximize control utility in a single control node and distributed control nodes respectively. The evaluation shows that Redactor always satisfies more control objectives in improving control consistency and utility compared with Athens [58] and static priority schemes.



(a) Single path forwarding



(b) Load balancing

Figure 3.1: Conflicts of control programs

3.2 Background

3.2.1 Problem Statement

Due to the independent implementation and execution of SDN control programs, decisions made by different control programs may violate each other. In Figure 3.1a, h_1 communicates with h_2 through path $s_1 \to s_2 \to s_5$ with a 7.5Mbps flow. To relieve load on switch s_2, a load balancing program *load-balance* would like to migrate a part of traffic to paths $s_1 \to s_3 \to s_5$ and $s_1 \to s_4 \to s_5$ in Figure 3.1b. Meanwhile, some other control programs also intend to process the flow, i.e., *energy-save*, *firewall*, *waypoint*, *rate-limit*. However, simultaneous processing may result in conflicts.

1) Action Conflicts: Control program *firewall* does not allow any traffic generated by h_1 travel through switch s_3, as this path is reserved for other traffic. Therefore, packets of flow $h_1 \to h_2$ are blocked by the firewall rule in s_3, while control programs *load-balance* and *waypoint* instruct the flow going through s_3. This kind of conflicts usually results in contradictory actions, and is obvious to recognize by checking the action fields of rules.

2) Consequent Conflicts: However, some consequent conflicts may be difficult to identify. These conflicts may lead to low performance, although conflicts are not

revealed in action fields of rules. Control program *rate-limit* restricts the available bandwidth of flow $h_1 \rightarrow h_2$ to 2Mbps on path $s_1 \rightarrow s_4 \rightarrow s_5$. Meanwhile, control program *energy-save* would turn a switch into sleep mode to save energy when it experiences low utilization. If these control programs act independently without noticing each other, *load-balance* instructs switch s_1 split the flow traffic equally to the three subpaths with 2.5Mbps. *energy-save* detects the low utilization of s_2 and makes it into sleep mode, while path $s_1 \rightarrow s_4 \rightarrow s_5$ only allows 2Mbps for the flow and $s_1 \rightarrow s_3 \rightarrow s_5$ blocks the flow. At this time, the actual available bandwidth for the flow is only 2Mbps through $s_1 \rightarrow s_4 \rightarrow s_5$, which is even less than 7.5Mbps with single path forwarding in Figure 3.1a.

In the control plane, each control program acts independently and aims at achieving its objective to optimize a sub-aspect of the network resources, e.g., maximizing available bandwidth, lowering energy consuming. If the controller could assign weights 5.5:2 for the paths $s_1 \rightarrow s_2 \rightarrow s_5$ and $s_1 \rightarrow s_4 \rightarrow s_5$, this control plan satisfies the maximum objectives of programs $\{$*load-balance*, *energy-save*, *firewall*, *rate-limit*$\}$. Meanwhile, the expected performance of the flow is also guaranteed to the most extent, as the maximum control objectives are satisfied. Therefore, to achieve the desired performance, the control decision *sol* should maximize the control utility $O_{cps}(sol)$, which satisfies the maximum number of objectives $\{O_{p_i}\}$ of the control programs $cps = \{p_i\}$.

$$O_{cps}(sol) = \max \sum_{p_i \in cps} O_{p_i}(sol) \qquad O_{p_i}(sol) = \begin{cases} 1 & sol \text{ satisfies } O_{p_i} \\ 0 & \text{otherwise} \end{cases}$$

As control programs are planning on the same network resources but running independently, it is usually impossible to generate a global optimal plan for all the control programs. If the controller could sense intents of control programs and coordinate them in the decision-making process, it would generate solutions satisfying the most objectives of control programs.

3.2.2 Related Work

Existing conflict resolution approaches usually check rules generated by control programs before applying in the network. Jin et al. [60] coordinate rules based on priorities. AuYoung et al. [58] and Mogul et al. [59] use a policy-evaluated approach to select a proposal to implement. Ferguson et al. [66] and Ferguson et al. [67] merge control requirements into a policy tree and resolve conflicts with user-defined conflict-resolution operators. However, as control programs run independently, the proposals are usually suboptimal and could not satisfy all the control objectives.

High level abstractions are used to express control programs for composition [54, 55, 56, 57], which pay more attention on composition than the decision consistency. PGA [50] abstracts intents of policies related to network endpoints, and uses a graph structure to detect and resolve policy conflicts. Due to the long composition latency, it pre-composes input graphs, which is inconvenient to integrate policies dynamically. To avoid the inaccuracy of coarse-grained coordination, finer-grained methods with state checking are proposed [24, 52]. However, for these state-based coordination, the large number of state variables results in high complexity and long latency.

Declarative languages are also used to implement network management and operation systems [63, 65, 68], which enable verification and prevent misconfiguration with network-wide reasoning. The consistency of declarative programs is little studied, while [64, 69] use simple fixed prioritization for conflict coordination.

3.3 Collaborative Control Programs

3.3.1 Declarative Control Program Composition

To avoid the suboptimum problem of afterwards rules checking, we prefer to reconcile control programs during the decision-making process. The controller is responsible for coordinating control programs to generate consistent configurations. Due to the diversity and complexity of imperative programming, it is impossible for the controller to understand every detail of control programs, so that control programs need to indicate their requirements and intents explicitly to the controller.

The essence of the network processing is to find appropriate forwarding paths for communications while obeying the concerned policies, e.g., going through a firewall or ensuring quality of service. Therefore, the objectives of control programs could be described as two aspects $O = \{Ns, Fr\}$: the desired state Ns of network resources (e.g., link utilization), and the resources demands Fr for flows (e.g., bandwidth requirement).

Instead of imperative programming, we use declarative languages to implement control programs, which do not need to care about the searching details to find solutions, but only express the logical objectives. A declarative control program

$$p(A, B, c, ...) : -rule_1(A, B), rule_2(B, c), ...$$

satisfies a series of logical rules $rule_1, rule_2, ...$, which are predicates or functions describing the desired network state Ns or flow requirements Fr objectives. The names of predicates, functions and constants begin with a lower-case letter, while variable names begin with an upper-case letter.

To express network processing with declarative languages, a flow from source S with port Sp to destination port Dp on D could be represented by a vector

$F = [S, D, Sp, Dp, ...]$ which consists of typical fields indicating a flow. A shortest path routing program $shortest(F, P)$ for flow F is defined with functions $path$ and $minimal$, in which $path$ searches all the available paths Set and $minimal$ finds the shortest $Path$ from the available paths.

$$shortest(F, Path) : -setof([P, L], (path(F.S, F.D, P), length(P, L)), Set),$$
$$Set = [\_|\_], minimal(Set, [Path, Len]).$$

$$path(S, D, P) : -travel(S, D, [S], Q), reverse(Q, P).$$

$$travel(S, D, R, [D|R]) : -link(S, D).$$

$$travel(S, D, V, R) : -link(S, A), A\backslash == D, \backslash + member(A, V), travel(A, D, [A|V], R).$$

Similarly, a minimum bandwidth guarantee program $mbg(F, P, Rb)$ is constructed with functions $path$ and $minbwcheck$ which checks whether a path P satisfies the minimum bandwidth demand Rb.

$$mbg(F, P, Rb) : -path(F.S, F.D, P), minbwcheck(P, Rb).$$

$$minbwcheck([X, Y], Rb) : -bw(X, Y, Ab), Ab > Rb.$$

$$minbwcheck([X, Y|T], Rb) : -bw(X, Y, Ab), Ab > Rb, minbwcheck([Y|T], Rb).$$

$$firewall(F, P) : -path(F.S, F.D, P), forall(member(A, P), firewallrule(F, A)).$$

With declarative control programs, the logics of control programs, configuration settings (e.g., firewall rules) and the network information (e.g., network topology, switch information, available bandwidth) are stored in a knowledge base. Thus, control program $firewall$ is able to check whether the flow violates $firewallrule$ stored in the knowledge base along routing path P. With the knowledge base, the behaviors of control programs are queries and modifications to the knowledge base. Any decisions made by control programs should follow logics and information in the knowledge base. Therefore, the common knowledge base enables the controller to ensure the decision consistency of control programs.

When a set of control programs $cps = \{p_1, p_2, ..., p_n\}$ would like to process a flow F simultaneously for various objectives, a consistent forwarding plan should meet all the objectives. For declarative programming, the program composition requires to combine the logics of programs. As control programs are stored in the same knowledge base, the composition could be achieved with the conjunction of programs with consistent inputs, by simply assembling the interfaces of control programs without reorganizing the logic details. Thus, the composition of control programs $p_1, p_2, ..., p_n$ is

$$cnj(V_1, ..., V_m) : -p_1(V_1, c_1), p_2(V_1, V_2), ..., p_n(V_2, V_m, c_1)$$

in which $V_i \in \bigcup p_j.vars$ $(i = 1, ..., m)$ are the variables of the participating control programs $p_j \in cps$, and c_i are the constant parameters of control programs. Similar to the variables coordinating rules in a program, variables are also used to coordinate control programs in the conjunction, e.g., V_1 in p_1 and p_2. Therefore, to find a suitable forwarding plan for flow f supporting $shortest$, mgb and $firewall$, we use conjunction $cnj(P) : -shortest(f, P), mgb(f, P, b), firewall(f, P)$ to compute path P for f satisfying firewall policies and bandwidth requirement b, in which f and b are known as constants.

The conjunction assembles control programs to be a multiple-objective program to produce solutions satisfying all previously independent objectives with once execution. The solutions of the composed program are also the conjunction of each control program's solutions $Sol(p_1, p_2, ..., p_n) = Sol(p_1) \cap Sol(p_2) \cap ... \cap Sol(p_n)$. If there is no conflict in the logics of the control programs, we can get feasible composed solutions $Sol(p_1, p_2, ..., p_n) \neq \varnothing$ which are also suitable solutions for each control program $Sol(p_1, p_2, ..., p_n) \subseteq Sol(p_i)(i = 1, ..., n)$. Otherwise, the conjunction of control programs could not generate any solution $Sol(p_1, p_2, ..., p_n) = \varnothing$ when conflicts exist among participating control programs, e.g., *waypoint* requires the flow going through s_3 while $firewall$ drops f on s_3 in Figure 3.1b. In the case of contradictions, the controller has to sacrifice some control programs to resolve conflicts and achieve the maximum control objectives.

3.3.2 Generating Control Program Composition

Different types of flows usually trigger diverse kinds of control programs, therefore, control program composition requires to be generated dynamically at run-time. The key to compose control programs is to coordinate requirements of control programs and then generate the correct conjunction.

3.3.2.1 Consistent Requirements

Before deriving the composition, the requirements of different control programs should be consistent. Firstly, we distinguish requirements from the objectives of control programs. Requirements are the input parameters specified by control programs exposed to the controller, and objectives are encoded and hidden in the logics of control programs which are invisible for the caller. For control programs $ratelimit(F, P, B) : -path(F.S, F.D, P), bwcheck(P, B)$ and $ratelimit(F, P) : -path(F.S, F.D, P), bwcheck(P, b)$ which both use predicate *bwcheck* to ensure the limitation, the bandwidth limitation B in $ratelimit(F, P, B)$ is a requirement specified by the control program, while $ratelimit(F, P)$ does not propose any bandwidth requirement in its parameters but uses constant b in logics as its objective. In the requirement checking, we only focus on requirements in parameters and do not deal with logic details which are left to reasoning in the knowledge base.

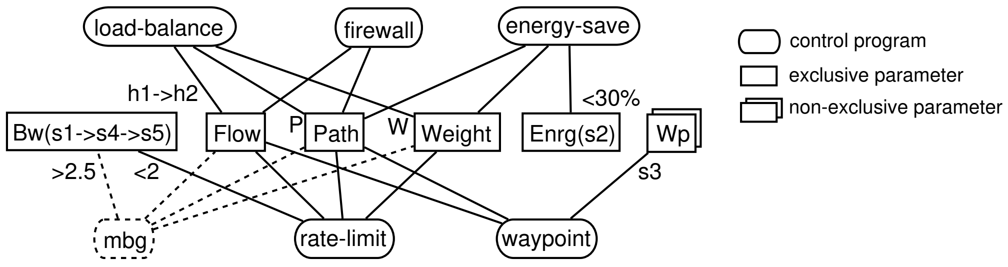


Figure 3.2: Requirement checking graph

We check the relationship between control programs and their requirements with Figure 3.2 for the control programs in Figure 3.1b. Each rectangle node indicates a potential parameter which is usually a resource item or variable to be calculated, and the oval nodes are the control programs. An edge is connected between a control program and a parameter node if the program requires to access the parameter, e.g., the parameters of *load-balance* are $Flow$, $Path$ and the corresponding $Weight$. Therefore, when a control program is newly added to the control node, it needs to define its concerned parameters. The value on each edge is specified by the control program for the connected parameter. As parameters of a control program consist of variables and constants, variables to be computed are indicated by upper-case letters without any instantiation, e.g., P and W in Figure 3.2.

With the requirement relationship graph, it is obvious that the control programs associated with a same parameter node should have a consistent view about it. For the nodes indicating resource requirements, e.g., bandwidth requirement $Bw(s_1 \to s_4 \to s_5)$, although control programs may propose different requirements for a same resource item, only one value is admitted, thus we call them *exclusive* nodes. For each *exclusive* parameter node N, the values on edges connected with N should be identical $\forall e_i, e_j = (N, *), l(e_i) \equiv l(e_j)$. In Figure 3.2, if a minimum bandwidth guarantee program *mbg* requests at least 2.5Mbps bandwidth for the flow on path

$s_1 \rightarrow s_4 \rightarrow s_5$, while *rate-limit* restricts it to 2Mbps, the requirement inconsistency could be detected as the values on edges between parameter node $Bw(s_1 \rightarrow s_4 \rightarrow s_5)$ with *mbg* and *rate-limit* are different. Contrarily, some nodes may admit multiple values, e.g., Wp allows multiple waypoint nodes for a flow. The inputs of these parameters do not need to be unified into a value.

3.3.2.2 Control Programs Conjunction

With the consistent inputs, most control programs could be composed directly to generate feasible solutions, and then the controller gets the corresponding result with the unified variable names.

However, the straightforward conjunction can not be applied directly when control programs make modifications to flows, e.g., changing the destination of flows for server load balancing. A flow modification program may deploy modification rules at any position A on its forwarding path, so that a flow is $f = [s, d, sp, dp, ...]$ on path $s \rightarrow ... \rightarrow A$ before arriving at A, and changed to $f' = [s', d', sp', dp', ...]$ on $A \rightarrow ... \rightarrow d'$. The modified flow should follow different policies between A and the destination from the original flow. Especially for the packet-level control programs (e.g., firewall) which check each packet of the flow, the modified packets are probably processed differently. Therefore, we treat the flow as two independent flows, i.e., f from s to A and f' from A to d', such that the packet-level control programs in the conjunction are divided for f and f' on subpath $s \rightarrow A$ and $A \rightarrow d'$ respectively:

$$p(f, s, d, P) \Rightarrow p(f, s, A, P_1), p(f', A, d', P_2), P = P_1 - A + P_2$$

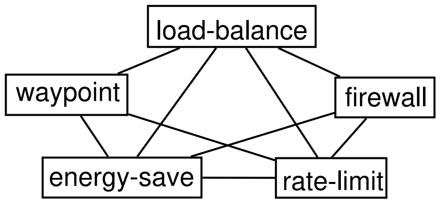
For flow-level control programs, e.g., routing, they only concern about reachability and performance of flows instead of modified details of packets. Therefore, modifications only affect the overall features (e.g., forwarding path P), and the control program does not need to process the original and modified flows separately with $p(f, s, d, P) \Rightarrow p(f \cup f', s, d', P)$.

3.3.3 Solution Selection

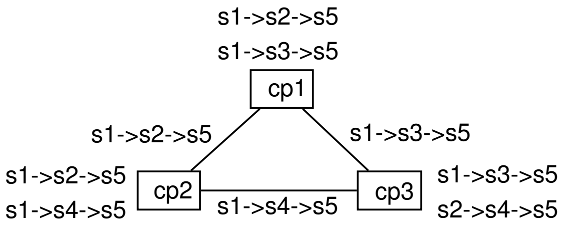
There are usually a lot of available solutions $Sol(p_1, p_2, ..., p_n)$ satisfying the control program composition, and we use Sol for simplicity. Nevertheless, these solutions may lead to diverse consequences with different performance when applied to the network. To achieve better network performance, we would like to select one with the best efficiency among the available solutions. However, control programs may prefer different solutions for their objectives. To reflect control programs' preferences, control program p defines its criteria $eval(p, sol)$ to evaluate solution sol. For example, the routing program would check the reachability and length of the routing path $eval(route, sol) = sol.get(P).reachable \cdot \frac{R}{sol.get(P).length}$ in which R is the radium of the network, so that shorter paths are preferred. Voting is a mechanism usually used for proposal selection [58, 59], which evaluates the potential efficiency of solutions. In this chapter, we use the cumulative voting scheme in [58], and choose the best solution based on the vote value of participating control programs $sol_{best} := \max\limits_{sol \in Sol} \sum\limits_{i=1}^{n} \frac{eval(p_i, sol)}{\sum\limits_{sol_j \in Sol} eval(p_i, sol_j)}$.

3.4 Conflict Coordination In A Control Node

The declarative control programs enable the coordination of multiple control programs to find a feasible solution. However, there may be conflicts among control programs, so that the participating control programs cps can not generate any feasible plan, resulting in an empty solution $Sol(p_1, p_2, ..., p_n) = \varnothing$. In this case, we would like to find a compromised solution with a subset of control programs



(a) Graph for Fig. 3.1b



(b) Complete graph without solution

Figure 3.3: Control programs conjunction relationship graph

$cps' \subseteq cps$ satisfying the most number of control programs, which also maximizes the control objectives $\max|\, cps' = \{p_i \mid \bigcap\limits_{p_i \in cps'} O_{p_i} \neq \varnothing\} \,|$.

To check the conjunction relationships of control programs, we construct a graph $G =< V, E >$ in which nodes V correspond to control programs and edges E indicate the conjunctions of each pair of control programs p_i and p_j. If $S(p_i, p_j) \neq \varnothing$, we add an undirected edge (p_i, p_j) in the graph. Intuitively, the nodes with more neighbours have higher probability to produce a solution in the control program composition. The subset cps' that could generate feasible solutions must form a complete graph, as each pair of control programs could always generate at least a feasible solution. Figure 3.3a shows the conjunction relationship of control programs in Figure 3.1b. Due to the conflict between *waypoint* and $firewall$, the maximum complete graphs consist of 4 control programs, i.g., {waypoint, load-balance, energy-save, rate-limit} and {firewall, load-balance, energy-save, rate-limit}.

However, a complete graph does not absolutely mean feasible composed solutions. For example, with Figure 3.1's topology, three control programs processing a flow $h_1 \rightarrow h_2$ form the complete graph showed in Figure 3.3b. Each control program generates two available solutions, e.g., cp_1 instructs the flow going through path $s_1 \rightarrow s_2 \rightarrow s_5$ or $s_1 \rightarrow s_3 \rightarrow s_5$. Although each pair of control programs has available solutions marked on the edges, the composition of these three programs derives no solution. Thus, the number of composed control programs that could generate feasible solutions is always no more than the largest complete graph size in the relationship graph. Therefore, the problem of compromised solutions is to find n' control programs $cps' \subseteq cps$ in the participating control program set cps satisfying

$$Sol(cps') \neq \varnothing, n' = \max |cps'|, 0 < n' \leq \max_{complete(G)} |G|$$

We design Algorithm 5 to find the best compromised solution, in which Line 2-7 construct the conjunction relationship graph and find all complete graphs. Control programs may be defined with different priorities, e.g., firewall programs have the highest priorities to ensure security properties of flows. For the complete graphs of the same size, in Figure 3.3a, we would like to choose control programs {firewall, load-balance, energy-save, rate-limit} with higher priority. Therefore, we have to deduce priority of each complete graph with the control programs in it and sort graphs based on the priority.

Priority comparison: We first sort complete graphs in descending order of graph size, and then compare priorities of complete graphs with the same graph size respectively. In each complete graph G, we sort control programs composing G in decreasing order of priority to be $G = \{p_1^G, p_2^G, ..., p_k^G\}$ with priority $r_1^G, r_2^G, ..., r_k^G$. For complete graphs G and G' with the same size k, if $\exists 0 \leq i \leq k$, such that $r_i^G < r_i^{G'}$ and $\forall 0 \leq j < i, r_j^G = r_j^{G'}$, we decide the priority $r(G)$ of G is lower than $r(G')$. Therefore, we can sort complete graphs with the same size based on the priority relationship.

Complete graph checking: To reduce the solution searching complexity, we check whether each complete graph could generate a feasible solution in descending

Algorithm 5 Conflict Coordination In A Control Node

```
1:  $CRG = \varnothing$ #control programs conjunction relationship graph
2:  for each pair $(p_i, p_j)$ in $cps = \{p_1, p_2, ..., p_n\}$ do
3:      if $Sol(p_i, p_j) \neq \varnothing$ then
4:          add edge $(p_i, p_j)$ in $CRG$
5:      end if
6:  end for
7:  $CG = \{G \mid complete(G), G \subseteq CRG\}$
8:  sort $CG$ according to graph size and priority
9:  $r = Priority.Lowest, cpcount = \max_{G \in CG} |G|$
10: $Soln = \varnothing$
11: while $Soln = \varnothing \wedge cpcount > 0$ do
12:     for $G \in CG, |G| == cpcount$ do
13:         if $Sol(G) \neq \varnothing \wedge r(G) \geq r$ then
14:             $Soln = Soln \cup Sol(G)$, $r = r(G)$
15:         end if
16:     end for
17:     $cpcount = cpcount - 1$
18: end while
19: return $sol_{best} = \max_{sol \in Soln} vote(sol)$
```

order of complete graph priority and size. We reduce the number of participating control programs *cpcount* gradually until finding feasible solutions with Line 9-18. For the complete graphs of the same size and priority, to get a solution with advanced performance, we generate all the feasible solutions and choose the solution with the highest vote value among these potential solutions to be the best in Line 19.

Complexity analysis: With n control programs processing a flow, the complexity for checking conjunction relationship of control programs is $O(\frac{n(n-1)}{2})$, and the complexity of conflict coordination is $O(|CG|)$ in the worst case when the complete graph checking could not find any feasible solution until checking all the complete graphs.

3.5 Consistency Coordination With Distributed Control Nodes

For networks with distributed control nodes, as these physically separated control nodes could not access logics of control programs deployed on other nodes, we can not compute the conjunction of control programs from different control nodes directly. Therefore, the coordination of control programs in distributed control nodes is more sophisticated.

For the requirement consistency, although requirements on each control node have been unified, demands among control nodes should also be consistent. We compare the parameter nodes in the relationship graphs of control programs on each control node and decide unified values for exclusive items before generating any solutions.

Each control node C_i $(i = 1, 2, ..., t)$ produces potential solutions $Sol(C_i) = \{sol_1^i, sol_2^i, ...\}$ independently. We use sol_i to simplify any solution in $Sol(C_i)$. A solution $sol_i = \{< V_x, rv_x > |V_x \in Var(C_i)\}$ of control node C_i consists of pairs of each variable V_x and its result value $rv_x = sol_i.get(V_x)$. To check the feasibility $C_j(sol_i)$ of solution sol_i on another control node C_j, we test sol_i with the control programs on C_j by replacing the undetermined variables with results in sol_i while the requirements are still specified by these control programs. $C_j(sol_i)$ is the control program composition on C_j, in which for all the variables $\forall V \in Var(C_j)$, if $\exists V \in Var(C_i)$, we substitute $V.value = sol_i.get(V)$, otherwise, $V.value$ keeps unchanged. If solution sol_i satisfies logics of all the control programs on all the control nodes $\forall C_j, C_j(sol_i) = true$, sol_i is a feasible solution maximizing the control consistency and utility. Even though control programs on different control nodes may focus on different aspects of network resources, the solution substitution checking ensures the global feasibility of solutions.

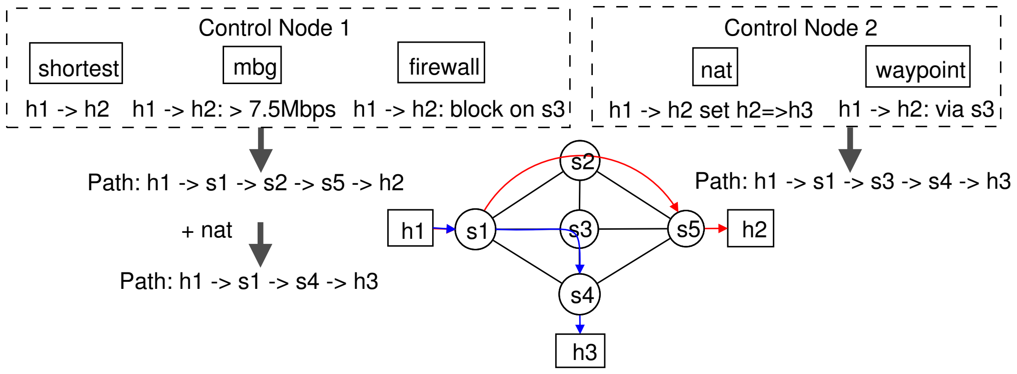


Figure 3.4: Control programs of multiple control nodes

However, a control node may make modifications to a flow, while other nodes process the flow straightforwardly without any changes. For the two control nodes in Figure 3.4, control node C_1 runs control programs {shortest, mbg, firewall} for a flow $h_1 \rightarrow h_2$, and C_2 applies a nat service on the flow by setting the destination h_2 to h_3 and the waypoint program requires the flow going through s_3. The control programs on C_1 find a route to h_2 satisfying their objectives, e.g., $h_1 \rightarrow s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow h_2$, while the destination of the flow on C_2 is h_3, so that the solutions of these two nodes could never satisfy each other. Therefore, it is impossible to find a consistent solution when control nodes support inconsistent views for flow modification.

We design Algorithm 6 to coordinate decisions of control programs on multiple control nodes. At the beginning, all the control nodes must make an agreement for flow modification with Line 1-8. In Figure 3.4, *nat* could be added to programs on C_1 to reproduce solutions, so that C_1 and C_2 both support flow modification. Line 3 regenerates solutions by applying modification to nodes previously without flow modification. There may be situations that applying the flow modification violates existing control programs on a control node, such that removing the flow modification from all the nodes would generate solutions supporting more objectives.

Algorithm 6 Coordination Of Multiple Control Nodes

```
1: if $\exists cps(C_i).has(mod) \wedge \exists \neg cps(C_j).has(mod)$ then
2:     for $\forall C_j, \neg cps(C_j).has(mod)$ do
3:         $Sol(C_j) = Sol(C_j) \cup Sol(cps(C_j) \cup mod)$
4:     end for
5:     for $\forall C_i, cps(C_i).has(mod)$ do
6:         $Sol(C_i) = Sol(C_i) \cup Sol(cps(C_i) - mod)$
7:     end for
8: end if
9: $Soln = \varnothing$
10: for each node $C_i$ do
11:     $Sol'(C_i) = \{sol | sol \in Sol(C_i), \forall j, C_j(sol) = true\}$
12:     $Soln = Soln \cup \{Sol'(C_i)\}$
13: end for
14: if $Soln == \varnothing$ then
15:     $\forall C_i, sol_i \in Sol(C_i), count(sol_i) = 0$
16:     for $\forall C_i, sol_i \in Sol(C_i), \forall C_j, p_j \in cps(C_j)$ do
17:         $count(sol_i) = count(sol_i) + (p_j(sol_i) == true)$
18:     end for
19:     $Soln = \{sol \mid \max_{sol \in \cup Sol(C_i)} count(sol)\}$
20: end if
21: return $sol_{best} = \max_{sol \in Soln} vote(sol)$
```

Line 6 calculates none flow modification solutions for nodes previously with modifications. Thus, we can select the best solution considering both applying and none modification situations.

It is also probably that we can not get a consistent solution due to the contradictory objectives of different control nodes (Line 14). In Figure 3.4, $firewall$ on C_1 blocks flows on switch s_3 while *waypoint* on C_2 requires flows to go through s_3. When no solution could fit all the control nodes, each solution *sol* always could not satisfy at least one control node $C_j(sol) = false$, which further means *sol* can not satisfy at least one control program $p_j(sol) = false$ on C_j.

$$Sol(C_1, ..., C_t) = \varnothing \rightarrow \forall sol \in \bigcup_{i=1}^{t} Sol(C_i), \exists C_j, C_j(sol) = false$$

$$\rightarrow \exists p_j \in cps(C_j), p_j(sol) = false$$

To generate consistent solutions, we have to sacrifice some control programs to make a compromised solution. We would like to choose a solution which could support the most objectives of control programs to maximize the control utility. We test solution *sol* with each control program p_j on control node C_j, if *sol* satisfies $p_j(sol) = true$, the supported control program count $count(sol)$ is increased in Line 17. The solutions with maximum $count(sol)$ that support the most control programs are probably the best solutions (Line 19). Among the solutions satisfying the most objectives, we choose the one with the highest vote value to be the best solution (Line 21).

The coordination among multiple control nodes selects the best solution satisfying the maximum number of control programs to achieve the highest control utility. When there is no flow modification, the satisfied number is always no more than the summation of objectives supported by each control node $O(sol_{best}) = \max\limits_{sol \in \bigcup Sol(C_i)} c(sol) \leqslant \sum\limits_{i=1}^{t} O_{C_i}(Sol(C_i))$. For situations applying flow modifications, the relationship between $O(sol_{best})$ and $\sum\limits_{i=1}^{t} O_{C_i}(Sol(C_i))$ is undetermined, as there may be control programs previously missed but now satisfied (e.g., *mgb*) due to the changing of flows and paths.

Complexity analysis: As each solution has to test its feasibility on all the control nodes, the complexity of control nodes coordination $O(N_{cps} \cdot N_{Sol})$ depends on the number of participating control programs N_{cps} and solutions N_{Sol}.

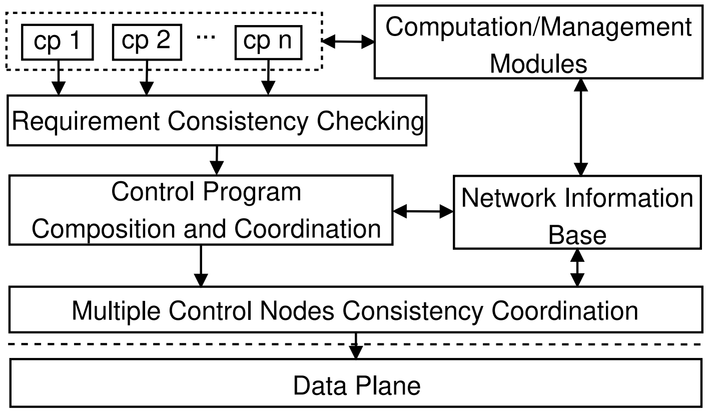

cp 1
cp 2
...
cp n
Computation/Management Modules
Requirement Consistency Checking
Control Program Composition and Coordination
Network Information Base
Multiple Control Nodes Consistency Coordination
Data Plane

Figure 3.5: Redactor architecture

3.6 Implementation

3.6.1 Architecture

We develop declarative control programs with the logic programming language Prolog, and these control programs could be integrated as modules with the Prolog engine in existing controllers, e.g., Floodlight, NOX. The architecture for control programs coordination is showed in Figure 3.5 implemented with 1000+ lines of Java codes. Each control node has a network information base (NIB) which stores network information and configurations, e.g., network topology, available bandwidth, and logics of programs. Declarative control programs search available solutions for their unknown variables with defined logics according to the information in NIB. Therefore, NIB acts as a database and control programs query NIB to find solutions that do not violate existing facts in the database. With Algorithm 5, control programs in the same control node produce rules consistently. While each control node generates decisions independently, for the control plane with distributed control nodes, Algorithm 6 coordinates proposals from control programs on multiple control nodes before deploying in the data plane.

3.6.2 Knowledge Query And Update

With NIB acting as a database, information (e.g., available bandwidth of links and utilization of switches) must be stored and updated in real time to make queries precise. Prolog provides *retract* function to remove existing knowledge and *assert* to insert new facts. Therefore, administrators could deploy new policies at run-time, e.g., security policies, by adding new facts in NIB. However, the facts to be inserted may conflict with the current knowledge base. The controller is responsible for checking whether facts to be inserted violate the knowledge base, and removes out-of-date facts to ensure no contradiction. Moreover, for the distributed control situation, network knowledge should be applied in all the control nodes consistently and concurrently. In this chapter, we assume the control coordination is based on the consistent knowledge bases on multiple control nodes, but do not address the NIB update problem for multiple nodes.

Comparing with control programs for policy checking (e.g., $firewall$) or ensuring service (e.g., *mbg*), routing programs (e.g., *shortest* in Section 3.3.1) have to reason available paths hop-by-hop according to the defined logics and topology information in NIB, which is a great overhead for a large-scale network. Considering the low performance of reasoning in Prolog, we relieve partial computation tasks to computation/management modules which are implemented by imperative languages. These modules compute and store available solutions as facts in NIB. Thus, the queries of declarative control programs to NIB just need to fetch the concerned information and select the facts satisfying specified conditions.

3.7 Evaluation

We evaluate Redactor with simulation in fat-tree networks and compare it with existing conflict resolution strategies. We implement eight types of declarative control programs: shortest path routing, load balancing, waypoint, firewall, nat, energy saving, minimum bandwidth guarantee, rate limit. The simulations are evaluated with 2-core 3.3GHz processor and 16GB RAM, and the network traffic is randomly generated and uniformly distributed among end nodes.

3.7.1 Declarative Control Programs Coordination

As declarative control programs always query logic and knowledge in NIB, the size and structure of NIB affect the performance of declarative programs. With the increase of network size, the size of NIB also grows quickly. Redactor could generate feasible solutions by composing control programs within tens of milliseconds for networks consisting of from 10 to 500 switches in Figure 3.6. Especially for the small-size networks with less than 200 switches, Redactor is able to find satisfiable solutions in less than 10 ms. As we have stored all the available paths in NIB, the query to NIB is searching paths satisfying objectives of control programs. The NIB query only accounts for a small portion of the total coordination time, which means the declarative searching is not a bottleneck for network control. For the larger fat-tree networks, there are more available paths for each flow. Therefore, the composition of control programs tends to generate more feasible solutions, and most of the coordination time is spent on solutions evaluation and selection.

To show the benefits of Redactor, we compare Redactor with SP (static priority prioritization), Athens-k (each control program generates k proposals), Redactor-sp and Redactor-vote which generate solutions with static prioritization and voting respectively when conflicts occur. Static priority always applies the rules with the

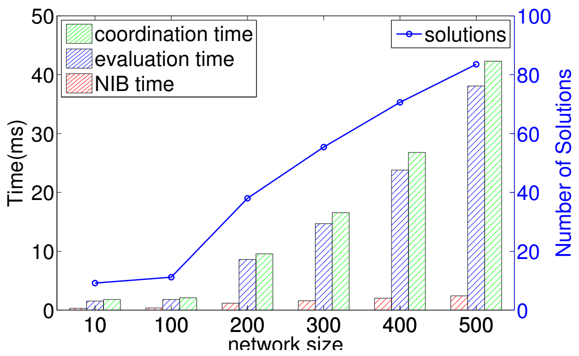


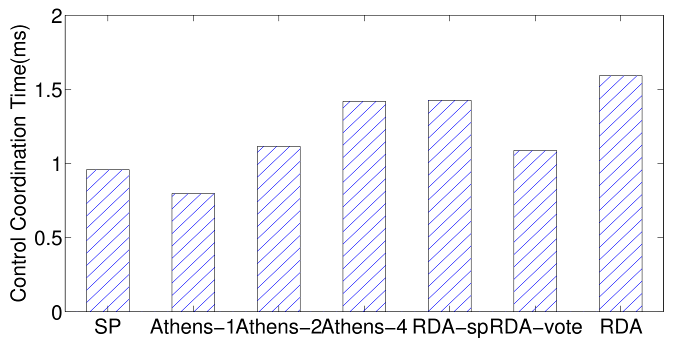
Figure 3.6: Control coordination with increasing network size

highest priority, while Athens generates independent proposals with each control program and determines the best solution with vote values of control programs.

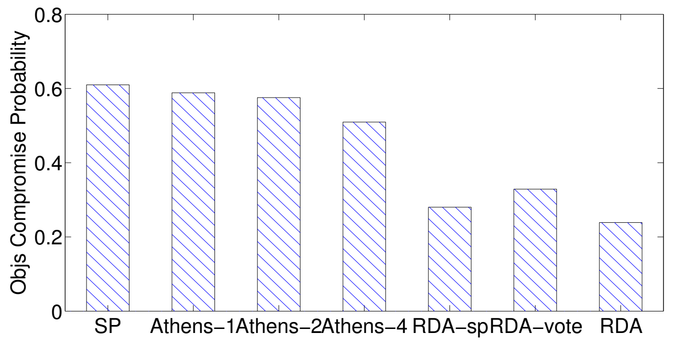
We evaluate these schemes in a fat-tree network with 100 switches. Although Redactor takes a little longer than other schemes in Figure 3.7a, Redactor selects the best solution among all the feasible plans generated by the composed control program, which avoids the suboptimal problem. During the control coordination, some control programs may be compromised to make control decisions consistent because of conflicts. Redactor maximizes the global control objectives and uses voting to select a solution with preferred network performance. Therefore, there is less probability of control program compromise and less number of compromised control programs with Redactor than other schemes in Figure 3.7b and 3.7c, which means Redactor achieves the best control utility.

3.7.2 Control Programs Coordination In A Single Node

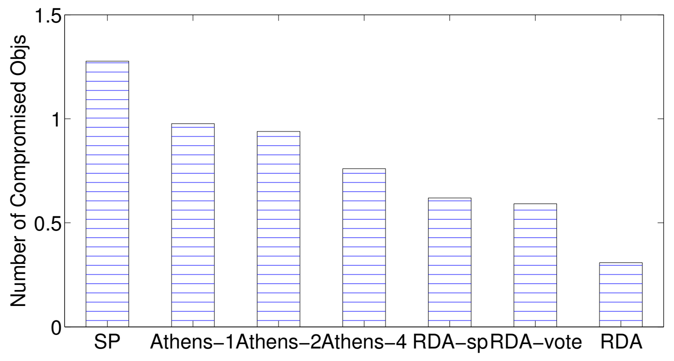
Various control programs are deployed in a single control node to configure the network. With the more number of deployed control programs, a flow is probably processed by more control programs, which tends to result in more control conflicts.



(a) Control coordination time



(b) Objectives compromise probability



(c) Compromised objectives number

Figure 3.7: Comparison with other coordination strategies

In this section, we evaluate how the number of participating control programs affects the control coordination.

Figure 3.8 shows the possibility of requirement inconsistency grows almost linearly when the number of participating control programs increases. As the variables

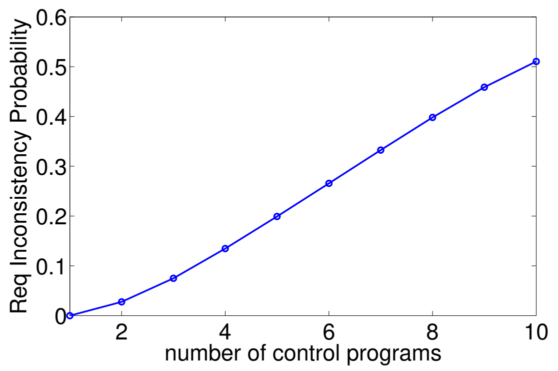
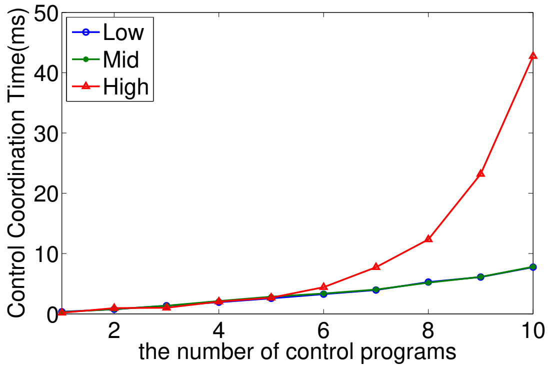


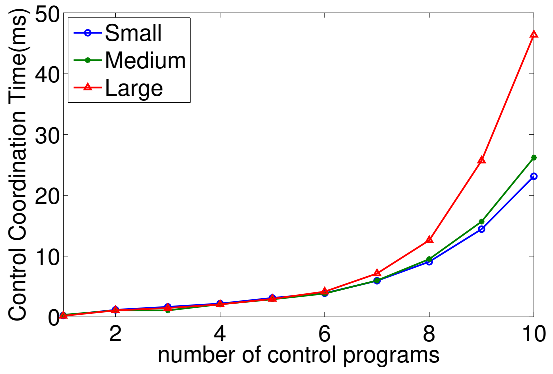
Figure 3.8: Requirement inconsistency probability

related to a flow are usually limited, the more control programs processing a flow at the same time, the higher possibility to access a variable related to the flow simultaneously. When 10 control programs are processing a flow, the probability of requirement inconsistency will be as large as 50%. Therefore, requirement consistency checking is essential for control programs coordination.

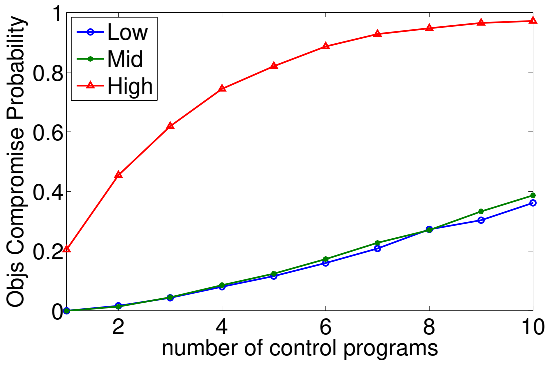
A lot of control programs are deployed to ensure quality of service for flows, e.g., minimum bandwidth guarantee, but these programs probably can not be satisfied under heavy traffic load. Therefore, the traffic load usually determines the control decisions, and the decision generation processes are also different under different traffic load. We evaluate Redactor in low (network utilization <30%), medium (network utilization 30%∼70%), and heavy (network utilization >70%) traffic load in Figure 3.9. The network utilization is measured by weighted link utilization. In Figure 3.9a, the control programs coordination under heavy network load always takes longer latency. Especially with more number of participating control programs, it could be as much as 5 times larger than the low and medium load situations, as it has to sacrifice a subset of control programs to regenerate a feasible solution satisfying the maximum objectives. Hence, the compromise possibility and the number



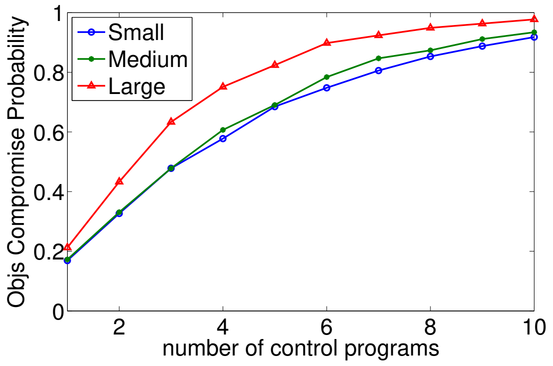
(a) Control coordination time



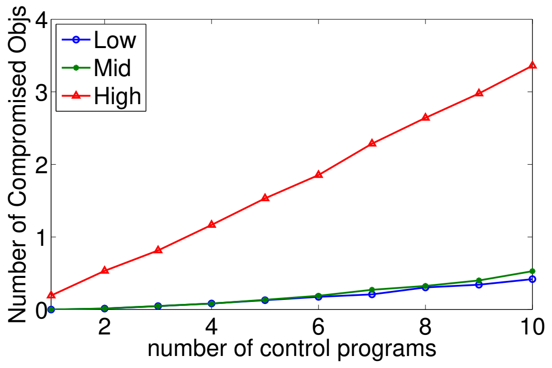
(a) Control coordination time



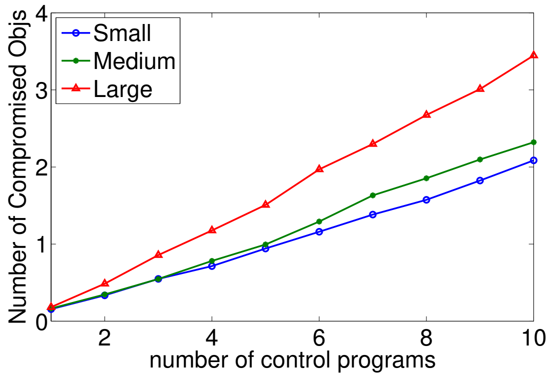
(b) Objectives compromise probability



(b) Objectives compromise probability



(c) Compromised objectives number



(c) Compromised objectives number

Figure 3.9: Control coordination under different traffic load in a single node

Figure 3.10: Control coordination with different flow size in a single node

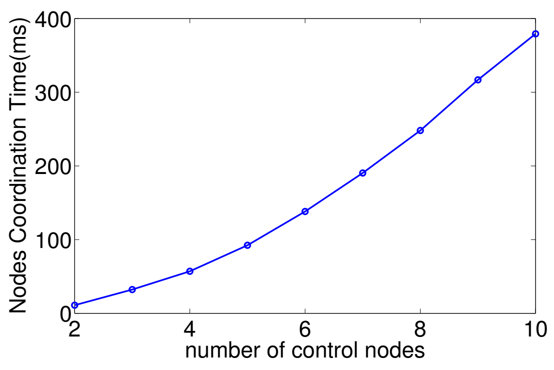
of compromised control programs under heavy load are significantly larger than the low and medium situations, in which the compromise possibility approaches 1 when 10 control programs are processing a flow simultaneously.

Meanwhile, the flow size also affects the solution generation process, especially under heavy network load, in which large flows probably could not be allocated with enough resources while small flows are satisfied with a small amount of resources. We experiment flows of different sizes (small flows <1M, medium flows 1∼10M and large flows >10M) under heavy network load in Figure 3.10. With less requirements for network resources, the small and medium flows have higher probability to be satisfied, while large flows tend to sacrifice more compromised objectives in Figure 3.10b and 3.10c. As Redactor searches compromised solutions with the decreasing number of participating control programs when conflicts occur, a large flow takes more rounds to find feasible solutions during the coordination. Hence, the coordination time for large flows is also longer than small and mediums flows in Figure 3.10a.

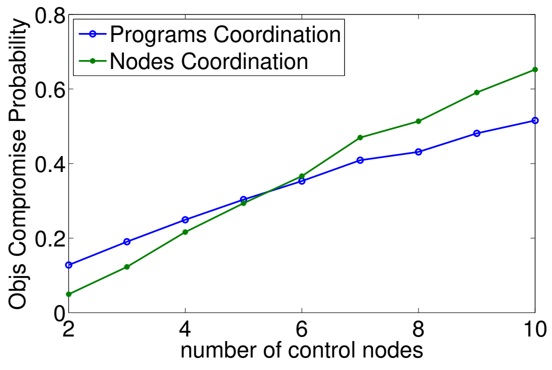
3.7.3 Multiple Control Nodes Coordination

In a logically centralized control plane with multiple control nodes, each control node runs several control programs independently. The increasing number of control nodes produces more solutions. As each solution has to test its feasibility with control programs on other control nodes, the time of control nodes coordination increases almost exponentially in Figure 3.11a when the number of control nodes grows, which demonstrates the complexity of multiple nodes coordination.

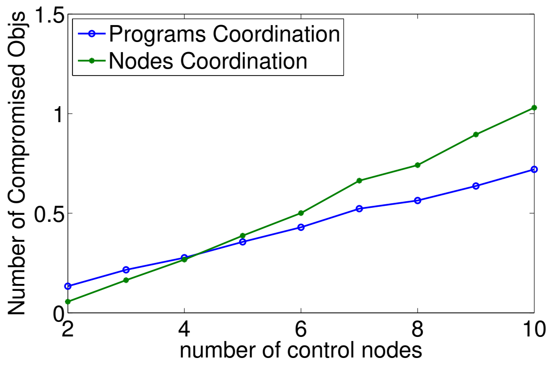
The control nodes coordination is expected to discover and resolve conflicts among control programs in distributed nodes, while decisions made by each control node are consistent for control programs on it. When the number of control nodes is small, most conflicts could be resolved locally inside each control node. As the more control nodes introduce more conflicts among distributed control programs,



(a) Nodes coordination time



(b) Objectives compromise probability



(c) Compromised objectives number

Figure 3.11: Control coordination with multiple control nodes

the control nodes coordination plays an important role in resolving conflicts among nodes. In Figure 3.11c and 3.11b, when the number of control nodes increases, the control nodes coordination resolves more conflicts with more compromised control programs. Even though the nodes coordination takes almost 10 times longer in Figure 3.11a than control programs coordination in a single control node, it is critical to ensure the global consistency. Nevertheless, the latency is still acceptable for a feasible solution with the best control utility.

3.8 Discussion

As the control reconciliation is based on the assumption that all the control programs are modelled with the declarative programming framework, the logic programming restricts the diversity and expressiveness of control programs, and poses challenges to the integration of imperative control programs. For instance, imperative programs could check and modify details in packets flexibly, while declarative programming is incapable to manipulate data outside the knowledge base. As the solution computation and coordination of declarative programs are all left to the compiler, another limitation is that the performance of logic reasoning strongly depends on the implementation of logic, which is usually slower than the imperative programs to find all the solutions satisfying the defined logic. To support both imperative programs and declarative programs, we would model the network information as a database, so that decisions made by imperative programs could also be verified with the database. The imperative programs are not involved in program composition during the decision-making process, and require an interface to query the feasibility of their decisions in the database. The interface program checks whether the decisions (e.g., forwarding path, allocated resource) made by imperative programs (e.g., C, Java) satisfy objectives of other declarative programs and current knowledge in the database. Meanwhile, the database is not limited for Prolog, and it could support other declarative languages, e.g., Datalog, SQL. To overcome the low efficiency of reasoning, we could relieve computation tasks to imperative programs, and logic programs are only responsible for searching and checking to ensure the consistency. Therefore, programs could be separated into computation and checking parts. For example, an imperative program calculates bandwidth guaranteed paths, and the logic checking ensures the paths generated by the corresponding imperative program or other control programs (e.g., routing, rate-limiting) indeed satisfy the required bandwidth demand. However, imperative control programs run independently without sensing the objectives of each other and probably lead to suboptimal results. Therefore, there is a trade-off for control consistency coordination.

3.9 Conclusion

Considering the increasing number and types of SDN control programs deployed by various control domains, we design Redactor to reconcile control programs to make consistent and efficient network control decisions. Redactor reconciles network control by implementing and composing control programs with declarative languages. We use a heuristic approach to resolve control conflicts, and then generate control decisions to maximize the control utility considering the priority and control objectives. Redactor ensures the control consistency and utility both in a single control node and distributed control nodes. The evaluation results show Redactor always achieves the maximum control utility compared with static prioritization and Athens, and the coordination overhead is acceptable.

CHAPTER 4
Boosting The Benefits Of Hybrid SDN

This chapter talks about coordinating the centralized SDN control with traditional network protocols in hybrid SDN which consists of both SDN switches and legacy switches. As legacy switches are controlled by distributed network protocols running insides them, they are out of the SDN control. Decisions made by the SDN controller should be consistent with the uncontrolled distributed protocols to avoid network misbehaviors. Moreover, distributed network protocols may restrict the flexibility of SDN, e.g., forbidding SDN's forwarding decision because of conflicting with distributed routing, which constrains the potentially achieved network performance. Therefore, an effective coordination between SDN control and traditional networking not only requires to maintain the consistency, but also needs to exert the flexibility of partial deployed SDN to improve the network performance. In this chapter, we enhance the SDN controllability and flexibility over the entire hybrid network by planning the placement of SDN switches and designing hybrid traffic engineering algorithms, while complying with the principles of traditional network protocols.

4.1 Introduction

The birth of SDN brings new opportunities and solutions to computer networks with its flexibility and programmability, especially in traffic engineering and network management. Today, more and more network administrators would like to upgrade their network infrastructures to support SDN technology for its benefits. The evolving of a traditional network towards a full SDN deployment is usually an incremental process, during which administrators have to upgrade each network infrastructure manually and design considerable control for the hybrid network. Thus, the network upgrade takes several days or even years to deploy Google's fully software-defined WAN [16]. Moreover, network operators would like to deploy SDN technology incrementally in order to build confidence in its reliability and familiarity with its operations [70, 71]. Therefore, the benefits of SDN should be manifest at the early stage of upgrade in the hybrid network, which would make SDN more appealing for adoption.

During the upgrade process, with the coexistence of legacy and SDN switches in the network, inappropriate deployment of SDN switches and inconsiderate design of hybrid control logic would result in a more complex network with severely degraded performance [72, 73]. Coordinating the centralized SDN control with distributed network protocols is critical to ensure the correctness of network processing and improve the network resource utilization. For example, in the hybrid network with SDN nodes A, E, G shown in Figure 4.1, packets of a flow $A \rightarrow F$ arrive at G through path $A \rightarrow B \rightarrow G$ or $A \rightarrow B \rightarrow E \rightarrow G$. Although SDN switch G could forward packets flexibly to any neighbour according to the software-defined control decisions, G is not allowed to send the flow to A and B to avoid loops, e.g., $A \rightarrow B \rightarrow G \rightarrow A$, $B \rightarrow G \rightarrow B$. Even though neither SDN control nor distributed routing generates any loop in its own decisions, the hybrid forwarding may lead to loops due to the isolated control domains. Therefore, a hybrid SDN has to maintain the forwarding consistency to guarantee the connectivity and stability of communication. To cope with the uncontrollable and inflexible distributed routing,

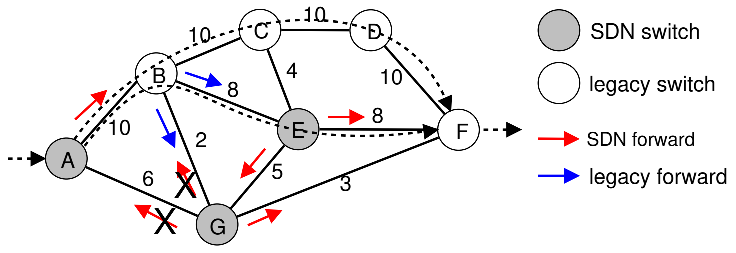


Figure 4.1: Forwarding of flow $A \rightarrow F$ in hybrid SDN: the number on each link is the available bandwidth, and the routing cost of each link is 1.

the flexible software-defined control should adapt to distributed decisions to avoid inconsistency.

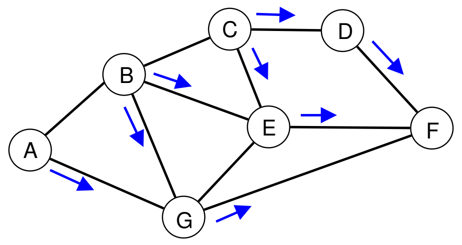
With partially SDN controlled network, the SDN controllability over the hybrid network varies with different deployed locations of SDN switches. To upgrade networks incrementally, existing approaches [30, 74] usually select the switches capturing the maximum traffic for upgrade, expecting most flows could be controlled by at least one SDN switch. However, the traffic pattern of a network is usually unknown and changes dynamically, so that it is hard to predict which switches are involved in maximum traffic. Moreover, when flows leave SDN switches, the SDN control over the flow is probably lost, as the downstream legacy switches may interrupt the desired forwarding path by choosing another next-hop. To extend the controllability, the placement planning of SDN nodes should consider the forwarding paths and forwarding characteristics of switches of different types. In addition, switches close to sources of flows are capable to control the access to the network, e.g., dropping blocked traffic based on rules in flow tables as early as possible to reduce resource consumption, so that these switches are expected to be upgraded to improve the access control. Therefore, the placement of SDN nodes requires considerate planning, otherwise the benefits of hybrid SDN will be limited.

To take advantage of SDN, it is essential to extend the flexibility of partially deployed SDN potentially over the entire network. Agarwal et al. [30] pioneer the traffic engineering in hybrid SDN, but they primarily focused on cooperating SDN control with single path routing on legacy switches. In traditional multipath routing, legacy switches would split traffic into multiple subpaths, which breaks the desired forwarding paths of SDN control. Due to the unsymmetric network topology and heterogeneous forwarding flexibility of switches, hybrid traffic engineering with legacy multipath routing is more complicated. In Figure 4.1, the next-hops of legacy switch B to F are E and G with equal-cost multipath routing. With the available bandwidth on links, we prefer flow $A \rightarrow F$ to take path $A \rightarrow B \rightarrow E \rightarrow F$ with 8 units bandwidth. However, B may split traffic equally into $B \rightarrow E$ and $B \rightarrow G$ with packet-level multipath routing. Unfortunately, the available bandwidth on link $B \rightarrow G$ is only 2 units, so that equally splitting on B will result in packet losses without network-wide information and optimization. To cooperate with both legacy single path and multipath routing protocols effectively, traffic engineering in hybrid SDN must take the forwarding capability of SDN switches and distributed routing into consideration. Furthermore, as most existing approaches adapt the flexible SDN forwarding to distributed routing to maintain forwarding consistency, the effectiveness of hybrid traffic engineering strongly depends on the next-hops and traffic splitting weights on SDN switches. However, existing hybrid traffic engineering approaches usually focus on the traffic splitting weights, ignoring the importance of forwarding next-hops, which probably restricts the potential effectiveness of hybrid traffic engineering.

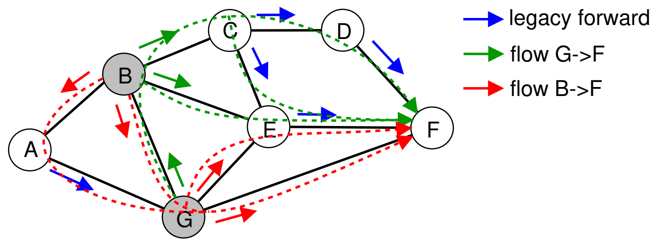
Moreover, the forwarding on legacy switches is limited to the least-cost paths calculated by distributed routing protocols of which the link metrics should be carefully designed. These least-cost paths restrict the flexibility of forwarding and potential performance of traffic engineering. Although path $A \to B \to C \to D \to F$ has more available bandwidth (10 units) than other paths in Figure 4.1, it is impossible to transport flow $A \to F$ with this path, as legacy switch B can not forward flows destined for F to C in the least-cost routing with current metrics setting.

In this chapter, we aim to design a generalized solution to handle the heterogeneity of different control domains in hybrid SDN, and therefore enhance the controllability and flexibility of the network with SDN placement planning and hybrid traffic engineering. We first check the property of hybrid forwarding considering the different control flexbility of SDN and traditional networking, and design the deployment positions of SDN switches based on potential hybrid forwarding paths. In SDN placement planning, we consider the controllability of software-defined control over the hybrid network, and design a heuristic method to incrementally select legacy switches for upgrade in order to achieve the maximum SDN controllability.

We then exploit the partially deployed SDN switches to control the traffic distribution to improve the network performance. First, we design a generic traffic engineering approach complying with the forwarding characteristics and capabilities of SDN and distributed routing, which supports both traditional single path and multipath routing protocols in legacy switches. To ensure the effectiveness of the hybrid traffic engineering, we also consider the influence of structures of hybrid forwarding graphs. Second, to overcome the limitation of distributed routing, we reconstruct forwarding paths for flows based on the proposed traffic engineering to achieve more forwarding flexibility. To apply the reconstructed paths, we use SDN controller to inject lies to control the distributed link-state routing protocols without modifying the implementation and configuration of legacy switches. The evaluation results with



(a) Legacy forwarding to F



(b) Hybrid forwarding to F

Figure 4.2: Forwarding paths to F

5 topologies show that our SDN placement planning guarantees more controllability over the network, and the hybrid traffic engineering ensures higher throughput and less packet losses especially in the early upgrade stage.

4.2 Hybrid Forwarding Graph

In traditional networks, legacy switches usually forward packets with the least-cost paths calculated by distributed routing protocols. These forwarding paths to the same destination could be constructed into a tree rooted with the destination node, e.g., the forwarding tree to F in Figure 4.2a. For a hybrid SDN $G = (V, E)$, in which switches $V = S \cup L$ consist of SDN nodes S and legacy nodes L, legacy nodes always forward packets based on next-hops on the least-cost paths, while SDN nodes could forward flows to their neighbours flexibly according to the decisions of software-defined control logic. Due to the flexibility of SDN nodes, flows from different sources to the same destination usually follow different forwarding rules. In Figure 4.2b with SDN switches B and G, switch G forwards flow $G \to F$ to B, while it can not send flow $B \to F$ which is received from $B \to G$ and $B \to A \to G$ back to B to avoid the loop. Therefore, the per-destination forwarding tree in traditional routing is insufficient to express forwarding of flows correctly in hybrid SDN. Although [30, 73] conduct traffic engineering with per-destination or per-source forwarding, we argue that:

Remark 1. *In hybrid SDN, flows $\forall s_i, s_i \rightarrow d$ to the same destination should follow different forwarding graphs $fg(s_i, d)$ in traffic engineering to achieve efficient forwarding.*

Definition 4. *Hybrid forwarding graph $fg(s, d)$:*

(1) $\forall n \in L, \forall m \in n.next(lc(n, d)), (n, m) \in fg(s, d)$

(2) $\forall n \in S, if\ (n, m) \in E \wedge cycle(fg(s, d), (n, m)) = true, (n, m) \notin fg(s, d)$

(3) $\forall n \in fg(s, d), n \neq d, fg(s, d).outdegree(n) > 0$

(4) $\forall n \in fg(s, d), n \neq s, fg(s, d).indegree(n) > 0$

To avoid inconsistency (e.g., loops, black-holes) in hybrid forwarding, we define a consistent forwarding graph for flow $s \rightarrow d$ with Definition 4. We keep next-hops $n.next(lc(n, d))$ of legacy switches on the least-cost paths $lc(n, d)$ to destination d with Definition 4(1), and ensure the outputs of SDN switches form no loop in the forwarding graph with Definition 4(2). Therefore, the hybrid forwarding graph is loop-free, as neither SDN nodes nor legacy switches forward packets into loops. Definition 4(3) guarantees the blackhole-free property as each node in the forwarding graph has at least an output port to forward the flow, and (4) gets rid of the irrelevant nodes and edges to make the forwarding graph concise. With hybrid forwarding graphs $fg(s, d)$, we can derive hybrid forwarding paths $P(fg(s, d), s, d)$ for flow $s \rightarrow d$ following the next-hops.

To be noted, different path discovery algorithms usually result in various hybrid forwarding graphs for a flow by applying distinct forwarding decisions on SDN switches following Definition 4. We will discuss the differences of these forwarding graphs for traffic engineering in Section 4.4.2.

4.3 Hybrid SDN Placement Planning

4.3.1 SDN Placement Planning

As the upgrade of a traditional network to a full SDN deployment is usually a long process spanning several months or even years, different SDN switches placement probably results in different controllability and flexibility with the software-defined control. Therefore, the positions of SDN switches in the hybrid network should be carefully designed to grasp the most controllability of the entire network. With the hybrid forwarding graphs in Section 4.2, we consider three factors in deploying SDN nodes during the upgrade process: traffic controllability, path controllability and access controllability.

4.3.1.1 Traffic Controllability

In practical network planning, traffic pattern and distribution are usually unknown without extra network measurements. In this case, we have to estimate the traffic that switches probably capture based on the hybrid forwarding characteristics.

The forwarding flexibility of legacy switches is limited compared to SDN nodes. To support both traditional single path and multipath routing protocols, we assume legacy switch n could forward $fp(n, m, s, d)$ percent of traffic $s \rightarrow d$ to next-hop m. For the single path routing, the ratio $fp(n, m, *, d)$ of legacy switch n forwarding to the exact single next-hop m is 1 for all the flows with destination d, while the possibility of forwarding traffic to the other nodes is 0. However, there are various next-hop selection algorithms in multipath routing. For instance, legacy switches usually use ECMP in multipath routing. ECMP decides the next-hop with hash-threshold algorithm [75], in which legacy switches perform a hash key (e.g., CRC16) over the packet header to decide the next-hop, while N next-hops have been assigned unique regions in the key space. $fp(n, m, d) = (m == \lfloor \frac{key}{regionsize} \rfloor)$ is calculated based on the hashing value and region design. Therefore, the forwarding of legacy switches depends on the distributed routing protocols, while the software-defined control algorithms determine the next-hops and forwarding ratio of SDN switches dynamically according to the network situation. We define that switch n is able to forward $tc(n, m, s, d)$ percentage of traffic $s \rightarrow d$ explicitly to next-hop m. As the forwarding percentage from a SDN switch n to m could be determined to be any value in $[0, 1]$, we use the maximum value 1 as $tc(n, m, s, d)$ on SDN switch n. tc reflects the forwarding flexibility from node n to node m.

$$tc(n, m, s, d) = \begin{cases} fp(n, m, s, d) & n \in L \\ 1 & n \in S \end{cases} \quad (n, m) \in fg(s, d) \tag{4.1}$$

The amount of traffic going through node n depends on the upstream paths from s to n on each forwarding path $p(s, n) \subseteq p \in P(fg(s, d), s, d)$. Thus, the traffic from s to d that SDN node n could control $TC(n, s, d)$ is defined as the summation of potential traffic on each subpath $p(s, n)$, which is the product of traffic forwarding flexibility of switch n_i to its next-hop n_j. $TC(n, s, d)$ indicates the percentage of traffic from s to d that SDN switch n potentially captures. We limit $TC(n, s, d)$ within $[0, 1]$, as we estimate the maximum amount of traffic going through each path with $tc(n, m, s, d) = 1$ on SDN switches, which would make TC larger than 1 on a downstream node.

$$TC(n,s,d) = \sum_{p(s,n)\subseteq p\in P(fg(s,d),s,d)} \prod_{(n_i,n_j)\in p(s,n)} tc(n_i,n_j,s,d) \tag{4.2}$$

$$TC(n,s,d) = \begin{cases} 1 & TC(s,d,n) > 1 \\ TC(n,s,d) & otherwise \end{cases} \tag{4.3}$$

In Figure 4.2b, for flow $G \to F$, B could capture all the traffic as G is able to forward the flow explicitly through link $G \to B$, such that $TC(B,G,F) = 1$. For switch D, although G and B could fully control the flow on its upstream subpath $G \to B \to C \to D$, legacy node C may forward the flow with next-hop E with ECMP which results in $TC(D,G,F) = 0$.

4.3.1.2 Path Controllability

SDN switches are able to manipulate and control the forwarding of traffic going through them, and the controllability should remain even if flows leave SDN nodes to ensure the effectiveness of SDN control. Therefore, the SDN deployment should ensure that the downstream paths of a SDN node could be controlled with few branches interrupted by legacy nodes.

A downstream path of node n for traffic $s \to d$ is $p(n,d) \subseteq p \in P(fg(s,d),s,d)$. We define the subpath controlled by SDN node n as $[n, n_1, ..., n_l] \subseteq p(n,d)$, in which if $n_j \in L$, $fg(s,d).outdegree(n_j) = 1$, so that the outgoing traffic of SDN nodes could follow the subpath explicitly without being redirected by legacy switches. The length of subpath controlled by node n on p is $pc(n,p) = l$, so that n is able to control $pc(n,p)/p.length$ of the path. Thus, the path controllability of SDN node n

for traffic $s \to d$ is the average path control among all the paths going through n.

$$PC(n, s, d) = \sum_{p \in P(fg(s,d),s,d)} \frac{pc(n, p)}{p.length} \cdot \frac{1}{|P(fg(s, d), s, d)|} \tag{4.4}$$

In Figure 4.2b, node B could control subpath $B \to E \to F$ for flow $G \to F$, while it can not fully control subpath $B \to C \to * \to F$ as the forwarding of C is out of SDN control.

4.3.1.3 Access Controllability

SDN switches are capable to control the access of flows to the network with rules installed by access control programs, e.g., dropping blocked packets, which is far more convenient than policy configruations on legacy switches. The denied packets should be dropped as early as possible to avoid resource consumption. To reflect the capability of SDN switches reacting to access control, we define access controllability $AC(n, s, d)$ which concerns positions of SDN nodes on hybrid forwarding paths.

The position of SDN node n on path $p \in P(fg(s, d), s, d)$ is $ac(n, p) = p.pos(n)$. Intuitively, the closer to the source, SDN nodes serve better access control for the flow. Therefore, the access controllability of n for traffic $s \to d$ is the average access control among all the paths going through n.

$$AC(n, s, d) = \sum_{p \in P(fg(s,d),s,d)} (1 - \frac{ac(n, p)}{p.length}) \cdot \frac{1}{|P(fg(s, d), s, d)|} \tag{4.5}$$

In Figure 4.2b, the accessibility of flow $G \to F$ could be determined at the first switch G which acts as a SDN node. Conversely, for flow $F \to G$ which always goes through link $F \to G$ directly, it almost arrives at the destination when G blocks it, while it is difficult to configure access control policies on legacy switch F.

The placement planning of SDN switches in hybrid SDN is to maximize the controllability and management of software-defined control over the network, such that it is able to achieve more benefits with the partial SDN deployment. Considering the path controllability and access controllability on the potentially captured traffic, the hybrid SDN upgrade planning task is to upgrade a set of switches ($|S| \leqslant k$) in appropriate positions to be SDN-enabled satisfying

$$\max \sum_{(s,d)} \sum_{n \in V} x_n(TC(n,s,d)(\alpha PC(n,s,d) + \beta AC(n,s,d))) \tag{4.6}$$

$$x_n = \begin{cases} 1 & n \in S \\ 0 & n \in L \end{cases} \qquad \sum_{n \in V} x_n \leqslant k \tag{4.7}$$

in which α, β are the weights tuning the focus of the controllability. For instance, if we set α and β to make $\alpha PC(n,s,d) + \beta AC(n,s,d)$ to be 1, it will upgrade switches capturing the maximum traffic. Thus, we provide a generalized model to formulate the SDN deployment problem to maximize the concerned controllability. It is a 0-1 integer programming problem by modelling type x_n of legacy switch n to be 0 and SDN switch to be 1, and it is known as a NP-complete problem [76]. In the initial network planning, it does not have strict time restrictions, so that the planning could be conducted by enumerating $C_{|L|}^{k}$ combinations to find the best SDN placement plan in the hybrid network.

4.3.2 Heuristic Upgrade Planning With Traffic Engineering

The purpose of upgrading a traditional network towards SDN is to provide better network service with the flexibility of SDN, e.g., high network utilization, fast failure recovery. For the online SDN placement planning in a running network, it should not only consider the network controllability but also serve the traffic demands.

To achieve better network performance, we use traffic engineering to optimize the network performance, e.g., minimizing the maximum link utilization μ (4.8). With the traffic distribution directed by traffic engineering, the traffic controllability $TC(n,s,d)$ of flow $s \to d$ on node n could be calculated with (4.9) in which $f(p)$ is the amount of traffic travelling through path p instead of the estimation in Section 4.3.1.1. (4.10) indicates the traffic demand $t(s,d)$ of flow $s \to d$ could be satisfied without any losses, and (4.11) means the occupied bandwidth on each link never exceeds the bandwidth capacity $c(e)$.

$$\max \frac{\sum\limits_{(s,d)} \sum\limits_{n \in V} x_n (TC(n,s,d)(\alpha PC(n,s,d) + \beta AC(n,s,d)))}{\mu} \tag{4.8}$$

$$TC(n,s,d) = \frac{\sum\limits_{n \in p \in P(fg(s,d),s,d)} f(p)}{t(s,d)} \quad \forall n \in S \;\; \forall s,d \tag{4.9}$$

$$\sum_{p \in P(fg(s,d),s,d)} f(p) \geqslant t(s,d) \quad \forall s,d \tag{4.10}$$

$$\sum_{e \in p \in \bigcup\limits_{\forall s,d} P(fg(s,d),s,d)} f(p) \leqslant \mu \cdot c(e) \quad \forall e \in E \tag{4.11}$$

$$x_n = \begin{cases} 1 & n \in S \\ 0 & n \in L \end{cases} \qquad \sum_{n \in V} x_n \leqslant k \tag{4.12}$$

For the online upgrade in a running network, administrators may expect to upgrade some legacy switches to be SDN switches quickly in a short period to serve better network performance for a traffic surge, e.g., upcoming Black Friday online shopping. In this case, choosing legacy switches for upgrade by enumeration in Section 4.3.1 along with traffic engineering seriously delays the upgrade process. We design a heuristic approach to select legacy switches for upgrade together with traffic engineering. We use $C(S, G, td) = \sum_{(s,d)} \sum_{n \in V} x_n(TC(n, s, d)(\alpha PC(n, s, d) + \beta AC(n, s, d)))$ to indicate the controllability of S over network G under traffic distribution td. We first compute the traffic distribution $td := \min \mu$ minimizing the maximum link utilization with current SDN deployment S, and then choose a legacy node to upgrade which achieves the best controllability $node := \max_{n \in L} C(S \cup n, G, td)$. Iteratively, we select nodes incrementally until achieving the desired number of SDN nodes. Compared to the enumeration planning, our heuristic approach searches only $|L|k - \frac{k(k-1)}{2}$ times, which significantly reduces the network planning overhead.

4.4 Dynamic Traffic Engineering In Hybrid SDN

With the partially deployed SDN nodes, we expect to utilize the flexibility of SDN to accommodate more traffic and reduce chances of congestions. The traffic distribution in hybrid SDN should be carefully designed to avoid performance degradation due to the distinct forwarding characteristics of SDN and legacy switches, especially when legacy switches are running multipath routing protocols, e.g., the multipath forwarding of B in Figure 4.1. In this section, we propose a generic traffic engineering approach by adapting the forwarding of SDN nodes to cooperate with legacy nodes in both single path and multipath routing scenarios.

4.4.1 Forwarding Capability Of Hybrid SDN

Although SDN nodes could forward traffic flexibly based on the decisions of software-defined control, traffic engineering has to satisfy both the forwarding characteristics of SDN and legacy switches. With the single path routing, legacy switches just need to forward a flow to the solo next-hop according to the destination. However, different multipath forwarding algorithms usually pose different restrictions to the forwarding of legacy switches by selecting different next-hops. We consider the flow-level and packet-level multipath forwarding on legacy switches, and describe their restrictions on forwarding with Definition 5.

Definition 5. *The amount of traffic $f(e, s, d)$ on link e for flow $s \to d$ should satisfy*

(1) $n \in L, \forall e_i = (n, n_i), e_j = (n, n_j) \in fg(s, d)$

for single path and flow-level multipath forwarding:

$\exists fp(n, n_i, s, d) = 1, f(e_i, s, d) \geqslant 0, \forall e_j \neq e_i, fp(n, n_j, s, d) = 0, f(e_j, s, d) = 0$

for packet-level multipath forwarding:

$$\frac{f(e_i, s, d)}{fp(n, n_i, s, d)} \equiv \frac{f(e_j, s, d)}{fp(n, n_j, s, d)}$$

(2) $\forall n \neq s, d, \sum_{e \in (*, n) \in fg(s,d)} f(e, s, d) \equiv \sum_{e' \in (n, *) \in fg(s,d)} f(e', s, d)$

Flow-level multipath forwarding: Legacy switches running flow-level multipath protocols forward a flow along a single path to reduce the likelihood of out-of-order arrivals, and different flows to a same destination may be forwarded to different next-hops on a legacy switch. ECMP is a flow-level multipath protocol, which forwards flows of the same IP source-destination pair along the same single path [77]. Flow-level multipath forwarding requires only one output link of a legacy switch could be used for a flow in Definition 5(1), which is similar to the single path forwarding.

Packet-level multipath forwarding: Packet-level multipath forwarding spreads packets of each flow along multiple least-cost paths, e.g., random and equal packet spraying algorithm RPS [78]. Chiesa et al. [77] and Dixit et al. [78] show that traffic splitting at packet-level granularity leads to significantly better load balancing in fat-tree topologies. To make it generic, the amount of traffic that a legacy switch n forwards to multiple output ports should be proportional to $fp(n, n_i, s, d)$ in Definition 5(1) which is decided by the multipath routing algorithm.

Switches should ensure no packet losses with Definition 5(2). These restrictions on legacy switches limit the amount of traffic that they could forward. Hence, we can infer the forwarding capability of each switch for a flow with Definition 6.

Definition 6. *The forwarding capability $c(n, s, d)$ of switch n for flow $s \to d$ is*

(1) $n = d, c(n, s, d) = +\infty$

(2) $n \in S, c(n, s, d) = \sum_{e=(n,n_i)\in fg(s,d)} \min\{c(n_i, s, d), u(e)\}$

(3) $n \in L$ *with single path or flow-level multipath forwarding:*

$$c(n, s, d) = \min_{e=(n,n_i),fp(n,n_i,s,d)=1} \{c(n_i, s, d), u(e)\}$$

$n \in L$ *with packet-level multipath forwarding:*

$$c(n, s, d) = \sum_{e=(n,n_i)\in fg(s,d)} fp(n, n_i, s, d) \cdot \min_{e=(n,n_j)\in fg(s,d)} \{\min\{c(n_j, s, d), u(e)\}/fp(n, n_j, s, d)\}$$

The forwarding capability of a switch is the maximum amount of traffic that it could forward, which depends on the available bandwidth $u(e)$ on output links $\{e\}$ and capability of downstream switches $\{n_i\}$. SDN switches are capable to forward traffic to any next-hops with the flexible software-defined control. Thus, the maximum amount of forwarded traffic is the summation of available throughput going through all the next-hops in Definition 6(2). For the legacy switches running flow-level multipath, the capability depends on the only next-hop. With packet-level multipath forwarding, the capability of a legacy switch is calculated by proportionally forwarding traffic to the next-hops while not violating the bandwidth restrictions in Definition 6(3).

With paths derived from the forwarding graph, we can get the forwarding capability of each switch. In Figure 4.1, the capability of B for flow $A \to F$ depends on $c(G, A, F) = 3$ and $c(E, A, F) = \min\{5, c(G, A, F)\} + 8 = 11$ with packet-level multipath forwarding. If the splitting ratio of the two next-hops $fp(B, E, A, F)$: $fp(B, G, A, F)$ is 1:1, the forwarding capability of legacy switch B is finally restricted by link $B \to G$ to be $c(B, A, F) = 4$. To be noted, the forwarding capability does not mean the exact amount of traffic the switch could forward to the destination. In Figure 4.1, if E forwards 3 units to G by assuming the capability of G is 3 units while B also sends 2 units to G, G is incapable to forward the 5 units. The capability only indicates the maximum amount of traffic that the node could process considering the forwarding characteristic and topology, which guides traffic engineering to avoid congestions.

4.4.2 Effectiveness Of Hybrid Traffic Engineering

Traffic engineering is usually formulated as a multi-commodity flow problem, e.g., [79, 80, 81], to satisfy the maximum fraction λ of traffic demands without violating the capability of links.

$$\max \lambda \tag{4.13}$$

$$\sum_{e \in p \in \bigcup\limits_{\forall s,d} P(fg(s,d),s,d)} f(p) \leqslant c(e) \quad \forall e \in E \tag{4.14}$$

$$\sum_{p \in P(fg(s,d),s,d)} f(p) \geqslant \lambda t(s,d) \quad \forall s, d \tag{4.15}$$

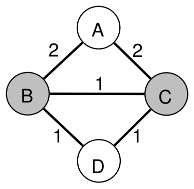
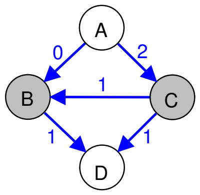
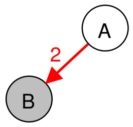


Figure 4.3: Topology

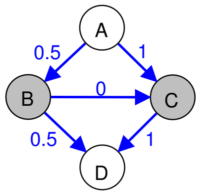


(a) flow $A \to D$

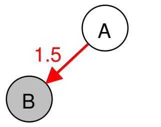


(b) flow $A \to B$

Figure 4.4: Forwarding scenario 1



(a) flow $A \to D$



(b) flow $A \to B$

Figure 4.5: Forwarding scenario 2

Figure 4.3 shows the network topology with bandwidth capacity of links, in which B and C are SDN switches while the other switches are legacy switches. Figure 4.4 and 4.5 are two different forwarding scenarios in which forwarding graphs for flows $A \to D$ and $A \to B$ (with throughput on links) are constructed respectively with different next-hop adapting approaches.

Thus, traffic engineering is conducted by scheduling traffic with available paths in the hybrid forwarding graphs. However, different path discovery algorithms probably result in distinct consistent hybrid forwarding graphs. Figure 4.4 and 4.5 show different forwarding graphs for flows $A \to D$ and $A \to B$ in topology shown in Figure 4.3. The forwarding graphs for flow $A \to B$ in the two scenarios are identical, while there is only a slight difference between the forwarding graphs of flow $A \to D$ on link $B \to C$ and $C \to B$. Even though these forwarding graphs all satisfy Definition 4, the slight difference in forwarding graphs would result in absolutely different consequences in traffic engineering. With the available paths in forwarding graphs, we can calculate the optimal traffic distribution with linear programming. In Figure 4.4, the network could accommodate 2 units for each flow with the forwarding graphs. However, Figure 4.5 shows that only 1.5 units are forwarded for each flow, which is 25% less than the amount of traffic in Figure 4.4. Although the traffic distributions in the two scenarios are both optimal with linear programming, the network performance varies due to the differences of the forwarding graphs. Therefore, the structure of forwarding graphs ultimately determines the effectiveness of hybrid traffic engineering.

To improve the potential optimal performance of traffic engineering, the forwarding graphs should be not only consistent but also effective in hybrid traffic engineering. As the next-hops of legacy switches are predetermined by distributed routing protocols, we only need to adjust the next-hops of SDN switches in forwarding graphs. We choose the forwarding graph with the maximum throughput for each flow to ensure potential large available bandwidth for flows.

4.4.3 Polynomial Time Approximation For Traffic Engineering

Although we can calculate the optimal traffic distribution for the two flows in Figure 4.4 and 4.5, computing optimal traffic distribution for a large number of concurrent flows is of great overhead together with frowarding restrictions on legacy switches. As the maximum concurrent flow problem is of NP-hard complexity, existing researches usually utilize approximation approaches for the dual problem that guarantee a solution within a logarithmic factor of the optimal solution, e.g., fully polynomial time approximation scheme (FPTAS) [82, 83, 84].

$$\min \sum_{e \in E} c(e) l(e) \tag{4.16}$$

$$\sum_{e \in p \in P(fg(s,d),s,d)} l(e) \geqslant z(s,d) \quad \forall e, l(e) \geqslant 0, \forall s,d \tag{4.17}$$

$$\sum_{(s,d)} t(s,d) z(s,d) \geqslant 1 \quad \forall s,d, z(s,d) \geqslant 0 \tag{4.18}$$

In the dual problem, $l(e)$ and $z(s,d)$ are introduced as dual variables, and $l(e)$ could be viewed as the routing cost of link e. The core idea of FPTAS is to forward flows

Algorithm 7 Hybrid Traffic Engineering

```
1: $\forall e, u(e) = c(e)$
2: for each flow $s \to d$ do
3:   $t(s,s,d) = t(s,d), \forall n \neq s, t(n,s,d) = 0$
4:   while $D(l) < 1 \wedge \exists n, t(n,s,d) > 0$ do
5:     $p(n,s,d) := \min_{p \in P(fg(s,d),n,d)} cost(p)$ with $\{l(e)\}$
           and $\forall (n_i, n_j) \in p, n_i \in L, fp(n_i,n_j,s,d) > 0$
6:     for $e = (n_i, n_j) : p(n,s,d)$ from last to first do
7:       $c = \min\{c, u(e)\}$
8:       if $n_i \in L$ then
9:         $c = \min\{\frac{c}{fp(n_i,n_j,s,d)}, c(n_i,s,d)\}$
10:      end if
11:    end for
12:    $c = \min\{t(n,s,d), c\}$, $t(n,s,d) = t(n,s,d) - c$
13:    for $e = (n_i, n_j) : p(n,s,d)$ do
14:      if $n_i \in L$ then
15:        for $\forall e' = (n_i, n'_j \neq n_j) \in fg(s,d)$ do
16:          $t(n'_j,s,d) = t(n'_j,s,d) + c \cdot fp(n_i,n'_j,s,d)$
17:          $u(e') = u(e') - c \cdot fp(n_i,n'_j,s,d)$
18:          $f(e',s,d) = f(e',s,d) + c \cdot fp(n_i,n'_j,s,d)$
19:        end for
20:        $c = c \cdot fp(n_i,n_j,s,d)$
21:      end if
22:      $u(e) = u(e) - c, f(e,s,d) = f(e,s,d) + c$
23:    end for
24:    $\forall e, l(e) = l(e)(1 + \epsilon\frac{\Delta f(e)}{c(e)})$
25:  end while
26: end for
```

with the least-cost paths calculated with $\{l(e)\}$ and update the cost $l(e)$ iteratively until exceeding the threshold $D(l) = \sum_{e \in E} c(e)l(e) < 1$.

Considering the forwarding restrictions and capability of switches in Definition 5 and 6, we design the forwarding Algorithm 7 based on FPTAS, such that the forwarding on SDN switches is adapted to maximize network throughput while complying with the forwarding characteristics of legacy switches. As FPTAS processes in phases until reaching the threshold, Algorithm 7 only shows a phase which consists of $|\{s \to d\}|$ iterations, and each iteration processes a flow in $\{s \to d\}$. Initially, the amount of traffic $s \to d$ on s to forward is $t(s, s, d) = t(s, d)$ (Line 3). For any other node $n \neq s$, as no traffic of flow $s \to d$ arrives, $t(n, s, d) = 0$. Similar to the general FPTAS approaches, we also choose the least-cost paths for flows. However, the least-cost path may contain legacy nodes of which the forwarding depends on the distributed routing protocols. We must ensure the availability of the path by checking the forwarding restrictions on legacy switches $\forall (n_i, n_j) \in p, n_i \in L, fp(n_i, n_j, s, d) > 0$ (Line 5). To determine the potential throughput along the path, we check the available bandwidth along the least-cost path and available downstream bandwidth of legacy nodes reversely from the destination to the source with Line 6-11. As legacy switches conduct traffic splitting with flow-level or packet-level multipath algorithms, the available downstream bandwidth depends on the splitting ratio and available bandwidth of downstream paths. To be noted, the available downstream bandwidth of legacy switch n does not equal the forwarding capacity $c(n, s, d)$, as the forwarding of SDN nodes along the selected path is determined and does not need the estimation with Definition 6(2). Thus, the amount of traffic could be forwarded from a legacy node n is the minimum value between $\frac{c}{fp(n_i, n_j, s, d)}$ which is restricted by the least-cost path and the forwarding capability of the node (Line 9). Line 13-23 forward traffic along the selected path, and divide the traffic among available next-hops with $c \cdot fp(n_i, n_j, s, d)$ when it meets legacy nodes. We do not forward the traffic directly to the destination along the branches at this time, but just forward it to the next-hop n'_j of each branch, so that the traffic to be transported $t(n'_j, s, d)$ on node n'_j is increased (Line 16). The traffic on branches will be forwarded in subsequent loops. In Figure 4.2b, when G sends out traffic c for flow $G \to F$ along

$G \to B \to C \to D \to F$, switch C forwards $c/2$ along the path to destination and another $c/2$ to E on the branch with equally packet-level splitting, so that $t(E, G, F)$ is increased by $c/2$ which will be processed later with paths from E to F in $fg(G, F)$.

With the amount of traffic $f(e, s, d)$ on each link for flow $s \to d$ in Algorithm 7, the traffic distribution among output ports on SDN node $n \in S$ could be determined with $td(e, s, d) = \frac{f(e,s,d)}{\sum\limits_{e,e' \in (n,*) \in fg(s,d)} f(e',s,d)}$.

An FPTAS guarantees that the solution has objective function value within $(1 + \epsilon)$-factor of the optimal for any $\epsilon > 0$ [30]. We set $\delta = \frac{1}{(1+\epsilon)^{\frac{1-\epsilon}{\epsilon}}}(\frac{1-\epsilon}{|E|})^{\frac{1}{\epsilon}}$ in Algorithm 7. Although Algorithm 7 considers the splitting characteristics of legacy switches in forwarding traffic, it still follows the general FPTAS procedure, so that the complexity is identical to [82] which is proved to be $O(\epsilon^{-2}|E|^2 log^{O(1)}|E|)$ and independent of the number of flows.

4.5 Path Reconstruction In Hybrid SDN

Even though the traffic engineering in hybrid SDN could leverage the flexibility of SDN to improve network performance, the forwarding on legacy nodes is still restricted to the paths calculated by distributed routing protocols, e.g, the incapability of B to forward packets destined for F to C in Figure 4.1. Thus, the forwarding flexibility and potential performance of hybrid SDN are still limited. If the routing calculation of distributed protocols in legacy switches could be controlled by software-defined algorithms, a more flexible forwarding scheme will be obtained. In traditional networks, this is usually conducted by either optimizing link weights [85] or using MPLS tunnels [86] to redirect traffic, which sophisticates distributed routing protocols or requires configurations on switches. With the partially deployed SDN switches, SDN is able to control distributed routing decisions by injecting lies [87] to control legacy switches running traditional link-state based protocols, e.g., OSPF or IS-IS, such that it enables flexible forwarding in both SDN switches and legacy switches. In this section, to achieve superior performance, we construct optimal paths with traffic engineering to break the restrictions of distributed routing, and redirect distributed routing by adapting the lies injection based approach in [87].

4.5.1 Paths Reconstruction With Traffic Engineering

To overcome the limitation of next-hops on legacy switches, we ignore the decisions of distributed routing and reconstruct next-hops of legacy switches to each destination based on traffic engineering. We consider an empty forwarding graph initially for each flow $s \to d$. Paths for a flow are constructed following the forwarding restriction and capability of switches and added into a graph, which form a reconstructed forwarding graph for each flow in Definition 7.

Definition 7. *Reconstructed forwarding graph $rfg(s, d)$*

(1) $\forall n \in L, e = (n, m) \in rfg(s, d), \forall s', e \in rfg(s', d)$

(2) $\forall n \in S, e = (n, m) \in rfg(s, d), e \in \exists p \in P(rfg(s, d), s, d)$

(3) $cycle(rfg(s, d)) = false$

For different flows $\forall s, s \to d$ to the same destination d, a legacy node should have identical next-hops for d in these reconstructed forwarding graphs $\forall s', rfg(s', d)$ with Definition 7(1). As the forwarding on SDN nodes is flexible, the edges associated with SDN nodes n are only established for the specific flow when a new path is reconstructed for it based on Definition 7(2). Definition 7(3) requires no loop in a reconstructed forwarding graph.

To achieve optimal forwarding, we reconstruct paths with traffic engineering in Algorithm 8. The least-cost path from node n to d for flow $s \to d$ is selected from all the available paths $P(G, n, d)$ in the topology G (Line 1). If the selected path does not exist in $rfg(s, d)$, and does not form any loop with the existing forwarding graphs, it is a newly constructed path. To avoid loops, we examine the selected path against $rfg(s, d)$, and check edges $(n, *)$ associated with legacy nodes $n \in L$ in other forwarding graphs $\forall s' \neq s, rfg(s', d)$ to the same destination d (Line 2). If the reconstructed edges do not generate loops in any $rfg(*, d)$, it is a feasible path. The edges associated with legacy switches and SDN switches are updated in concerned reconstructed forwarding graphs (Line 3-6) based on (1) and (2) in Definition 7. Otherwise, we keep on searching the least-cost paths despite the paths that have been checked until finding out a feasible path.

Lemma 3. *Flow $s \rightarrow d$ has at least one feasible reconstructed path in $rfg(s, d)$ as long as s and d are reachable in G.*

Proof. If there is no path reconstructed for $s \rightarrow d$, each path $p \in P(G, s, d)$ generated from G forms loops with at least a $rfg(*, d)$. If $n \in cycle(p, rfg(s', d))$ is the first node along $n \in p$ that forms the loop, $rfg(s', d).indegree(n) > 0$. As edge $(*, n) \in rfg(s', d)$ is established by a flow $s'' \rightarrow d$, there must be a path $n \rightarrow n''... \rightarrow d \in rfg(s'', d)$. If $n \rightarrow n''... \rightarrow d$ does not form loop with $rfg(s, d)$, and path $s \rightarrow ... \rightarrow n \rightarrow n''... \rightarrow d$ is a feasible path for $s \rightarrow d$ in $rfg(s, d)$. Otherwise, for the loop lp formed by $n \rightarrow n''... \rightarrow d$ and $rfg(s, d)$, there must be an edge $(n_i, n_j) \in lp, n_i \in S$, otherwise $lp \in rfg(s'', d)$. According to (2) in Definition 7, $\exists p^*, n_i \in p^*$, and p^* is a feasible path in $rfg(s, d)$. $\square$

We use the paths reconstructed by Algorithm 8 to replace Line 5 in Algorithm 7, so that the hybrid traffic engineering could utilize the reconstructed paths to achieve more forwarding flexibility.

Algorithm 8 Forwarding Path Reconstruction

```
1: $p(n, s, d) := \min_{p \in P(G,n,d)} cost(p)$ with $\{l(e)\}$
2: **if** $p(n, s, d) \notin rfg(s, d) \wedge \neg cycle(p(n, s, d), rfg(s, d)) \wedge \forall s' \neq s, \neg cycle(\{(n_i, n_j) \in p \mid \forall n_i \in L\}, rfg(s', d))$ **then**
3:     **for** each $e = (n_i, n_j) \in p(n, s, d)$ **do**
4:         if $n_i \in S$, then $rfg(s, d) = rfg(s, d) \cup e$
5:         if $n_i \in L$, then $\forall s', rfg(s', d) = rfg(s', d) \cup e$
6:     **end for**
7: **end if**
```

4.5.2 Apply Reconstructed Path With Fake Nodes

The topology and routing costs of links determine routing path calculation in legacy switches, but have no influence on SDN nodes. Moreover, SDN nodes differentiate the forwarding for different flows, while the next-hops of a legacy switch to the same destination in $\forall s, rfg(s, d)$ are identical. Therefore, to apply the reconstructed paths, we ignore the forwarding of SDN nodes and only enforce the reconstructed next-hops of legacy nodes. We consider the reconstructed forwarding graph $rfg(d)$ of all the legacy switches to destination d, e.g., reconstructed next-hops of legacy switches A, C, D, E to F in Figure 4.6. New next-hops $A \rightarrow B$ and $C \rightarrow B$ are introduced for A and C respectively in Figure 4.6, which are unavailable in Figure 4.2a. To enforce the reconstructed paths, we adapt Fibbing [87] to augment the topology from G to G'. Fibbing [87] coaxes legacy switches into computing forwarding decisions by presenting them with a carefully constructed augmented topology that includes fake nodes and fake links (with link cost), and applies augmented topology by injecting lies into link-state routing. The reconstructed next-hops in Figure 4.6 could be enforced by attaching fake destination nodes (i.e., F' which is a fake node for the destination in the subnet of F) to the next-hop changing switches (i.e., A, C) and designing the costs between the switches and fake nodes. For instance, the

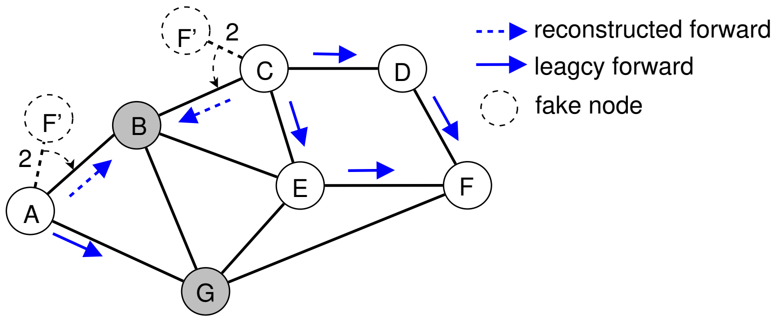


Figure 4.6: Augmented topology for traffic to F: the routing cost of each real link is 1, and the costs between the fake nodes and switches are shown on the fake link.

fake node is inserted between A and B with cost 2. Thus, A has two shortest paths $A \rightarrow G \rightarrow F$ and $A \rightarrow F'$ both with cost 2 to reach the destination in subset of F. Since the fake nodes do not really exist, packets forwarded to the fake node by A actually flow through B.

Fibbing [87] reduces the augmented topology with globally-scope lies which are visible to all the switches instead of locally-scoped lies for each switch. The fake nodes generated by globally-scope lies must be consistent to redirect the legacy forwarding correctly. However, due to the absence of SDN nodes' next-hops in the reconstructed forwarding graph, adding fake nodes only to the legacy switches with next-hop changes is insufficient to ensure the correctness. In the reconstructed forwarding graph to O with SDN nodes I and J depicted in Figure 4.7, we add fake nodes to legacy switches C, E, H, K, M which change their next-hops according to the augmentation algorithm in Fibbing [87]. As downstream switches with fake nodes probably attract traffic from an upstream switch without fake node, e.g., A with unchanged next-hops, because the cost going through a fake node is usually lower than original path to redirect forwarding, the fake node connected with C attracts traffic on A going through C to reach O, while another subpath $A \rightarrow D \rightarrow F \rightarrow J$ has

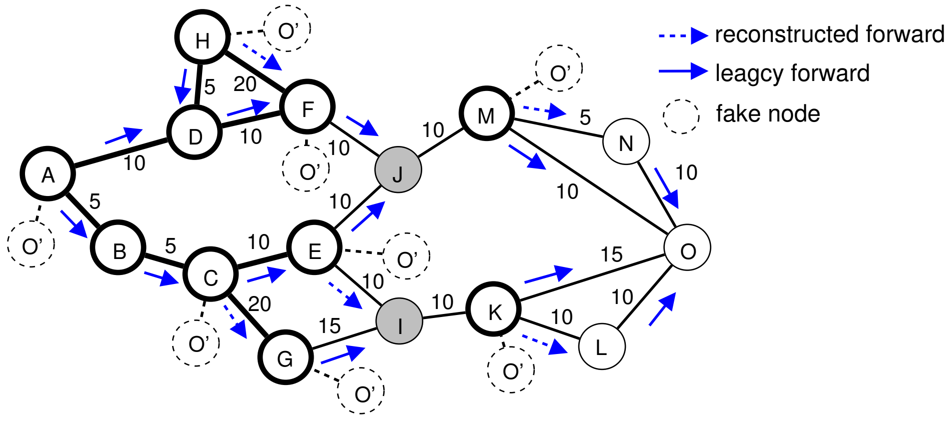


Figure 4.7: Reconstructed paths and fake nodes to O

no fake node to attract the traffic. Although there is a fake node connected with M on the downstream path, it does not attract A since SDN node J isolates them without any forwarding. Thus, the only next-hop of A for O is B instead of B and D, which violates the desired forwarding. To deal with the isolation problem in hybrid SDN, we reinvestigate the fake nodes injection for the reconstructed forwarding graph without next-hops of SDN nodes. We identify three kinds of switches to attach fake nodes in reconstructed forwarding graph $rfg(d)$ for destination d:

1) For the legacy switches with newly constructed next-hops $rn(d) = \{n \mid n \in L, \{n' \mid (n, n') \in rfg(d)\} \neq \{n' \mid (n, n') \in fg(d)\}\}$, e.g., C, E, H, K, M, we attach a fake node $fk(n, d)$ to each switch $n \in rn(d)$ to redirect the traffic.

2) To avoid potential isolations of SDN nodes, we add fake nodes to the legacy switches connected with SDN nodes $sn(d) = \{n \mid \exists n' \in G.neighbour(n), n' \in S, n \in L\}$, e.g., F, G.

3) The fake nodes connect to $rn(d) \cup sn(d)$ may affect the forwarding of upstream switches, e.g., the fake node connected with C attracts traffic from A and B. Therefore, we can generate an affected forwarding tree $aft(d) = \{n, e \mid p \in rfg(d), p.end \in$

$rn(d) \cup sn(d), e = (n, *) \in p\}$ in which the forwarding of switches depends on fake nodes connected to $rn(d) \cup sn(d)$, e.g., $A \rightarrow B \rightarrow C \rightarrow E/G$, $A \rightarrow D \rightarrow F$, $H \rightarrow (D) \rightarrow F$. To avoid a large number of fake nodes, we only add fake nodes to the switches with multiple next-hops $mn(d) = \{n \mid rfg(d).outdegree(n) > 1, n \in aft(d)\}$, e.g., A, to prevent potential imbalanced attractions of downstream fake nodes. For a switch with only one next-hop, the first downstream fake node is responsible for ensuring its correct forwarding, e.g, the fake node connected with C attracts traffic on B destined for O.

Therefore, the fake nodes are attached to real switches $fn(d) = rn(d) \cup sn(d) \cup mn(d)$ in augmented topology G', and each fake node $fk(n, d)$ connected to $n \in fn(d)$ is responsible for controlling the forwarding of the closest upstream switches without fake nodes $cn(fk(n, d)) = \{n_i \mid n_i \in p \in aft(d), p.end = n, p \cap fn(d) = n\}$.

To apply the reconstructed next-hops correctly, we adapt the bound propagation and merging algorithms in Fibbing [87] to calculate the routing costs between switches and fake nodes in G'. As each fake node $fk(n, d)$ controls the forwarding of a bunch of switches $cn(fk(n, d))$, the cost between n and $fk(n, d)$ ensures the correct forwarding of controlled switches with:

1) attracting traffic on controlled switches $n_i \in cn(fk(n, d))$ with $cost(n, fk(n, d), G') < \min\limits_{n_i \in cn(fk(n,d))} dist(n_i, d, G) - dist(n_i, n, G')$.

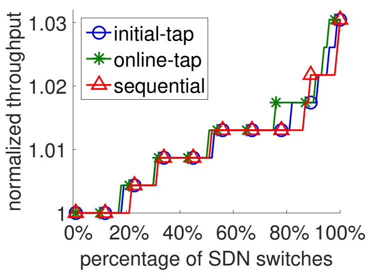
2) never attracting traffic on the other switches $n_i \notin cn(fk(n, d))$ with $dist(n_i, n, G') + cost(n, fk(n, d), G') > dist(n_i, n', G') + cost(n', fk(n', d), G')$ or $dist(n_i, d, G')$ for n_i controlled by $fk(n', d)$ or not controlled by any fake node respectively.

4.6 Evaluation

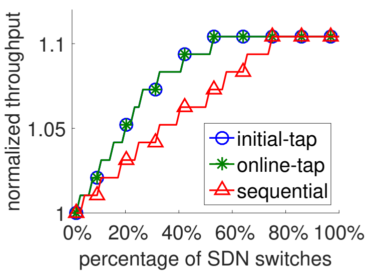
To show the effectiveness of the proposed SDN placement planning and traffic engineering approaches, we evaluate them in various topologies with simulation: Fat-tree (100 switches, 3 layers) and Mesh (64 switches with degree 4 for each switch) topologies which are widely used in data center networks and local area networks respectively, and three real-world ISP topologies in [88]: Telstra (108 nodes, 153 links), Ebone (87 nodes, 161 links), Exodus (79 nodes, 147 links). The traffic is randomly generated and uniformly distributed among nodes. As the traffic engineering with hybrid forwarding depends on the structure of forwarding graphs, different path discovery algorithms would result in diverse forwarding graphs. To avoid inconsistency and achieve high effectiveness, we accommodate the forwarding of upgraded SDN switches to the existing forwarding graph during the incremental upgrade process as Section 4.4.2 shows, so that the SDN placement planning plays an important role in composing forwarding graphs for traffic engineering. The simulations are evaluated with 2-core 3.3GHz processor and 16GB RAM.

4.6.1 SDN Placement Planning

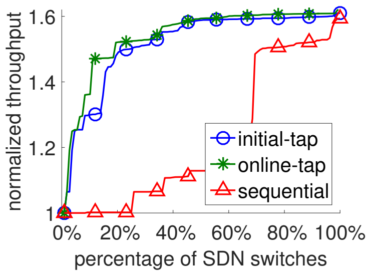
SDN nodes deployed by different placement strategies usually result in diverse controllability and flexibility over the network. As our SDN placement planning (named tap) is a generic approach, it could achieve concerned controllability by tuning weights α and β. To show the general effectiveness of tap, we set α and β to 1, and compare it with the sequential approach which upgrades switches in topological order. To evaluate the robustness of tap, we test both the initial hybrid network planning without traffic pattern (initial-tap) and online upgrade planning with traffic engineering (online-tap). The traffic controllability could be reflected by the performance of traffic engineering. Figure 4.8 shows the throughput improvement



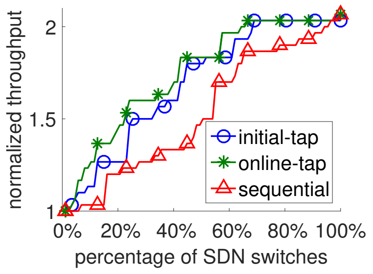
(a) Fattree



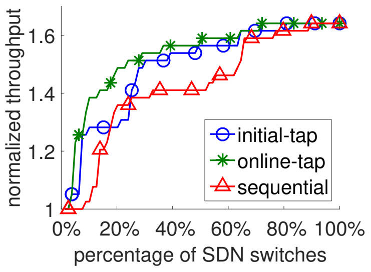
(b) Mesh



(c) Telstra



(d) Ebone

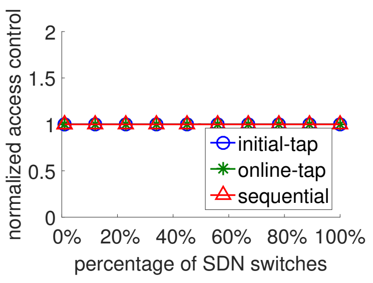


(e) Exodus

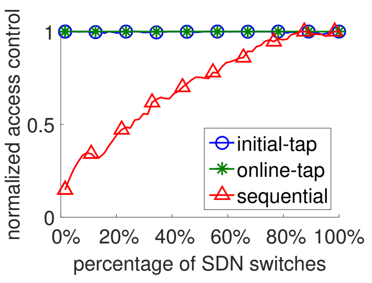
Figure 4.8: Throughput with different SDN placement strategies

of hybrid packet-level multipath forwarding during upgrading process with different upgrade strategies compared to traditional multipath routing $\frac{T(multiHyb-Pkt,SDN\%)}{T(multiHyb-Pkt,0\%)}$. Online-tap always achieves the most throughput, and the throughput of initial-tap is slightly less than online-tap, which demonstrates the robustness of the placement planning without traffic pattern. The situation of sequential upgrade is much worse due to the poor placement of SDN nodes. Only for Fat-tree network, the sequential approach first upgrades core switches, aggregate switches and then edge switches, such that it is similar to the planning considering traffic controllability.

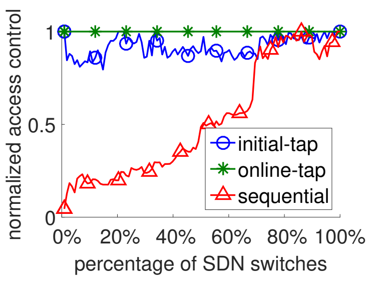
To evaluate the access control of SDN, we normalize the access controllability (Section 4.3.1.3) of these placement planning approaches with the online-tap $\frac{AC(planning,SDN\%)}{AC(online-tap,SDN\%)}$ in Figure 4.9. We find that the access controllability of online-tap and initial-tap is always superior than the sequential upgrade, which means the hybrid SDN requires a considerate upgrade planning to control the access of traffic.



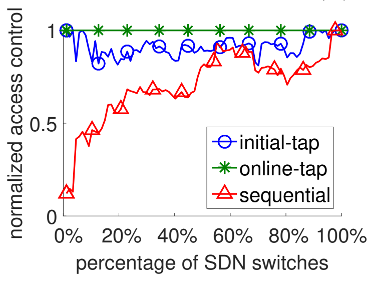
(a) Fattree



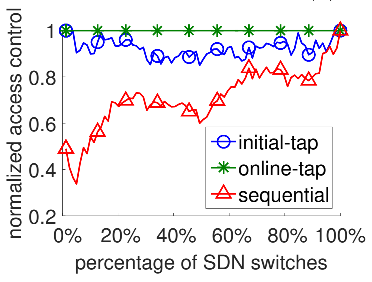
(b) Mesh



(c) Telstra



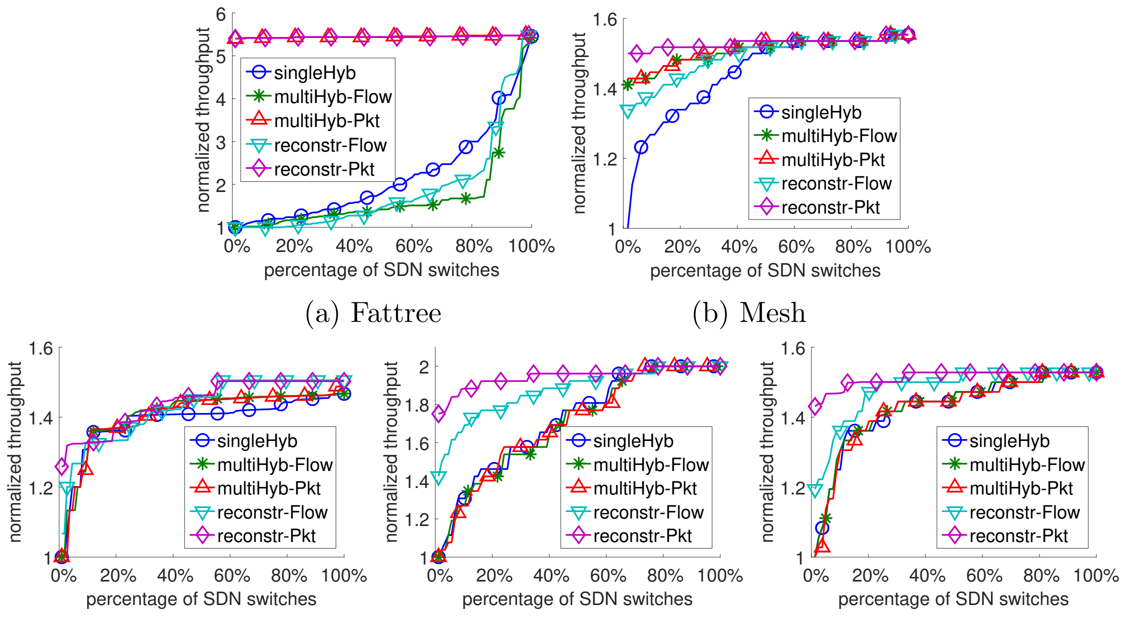
(d) Ebone



(e) Exodus

Figure 4.9: Access control with different SDN placement strategies

Although the traffic pattern is unknown in initial-tap, it achieves high access controllability which is similar to online-tap, as it considers access control over all the potential traffic. Therefore, our placement approach initial-tap could still plan the SDN placement for a dynamic network even if the traffic pattern is unknown. To be noted, the sequential upgrade from core switches to edge switches in Fat-tree could achieve almost the same controllability over the traffic. As we set α, β to be 0.5, and the path controllability concerns the downstream paths while the access controllability is determined by the upstream paths, tap is similar to only considering traffic controllability with the symmetric topology and uniform traffic in Fat-tree, in which the core and aggregate switches capture more traffic than edge switches.



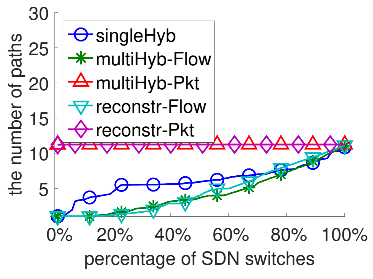
(a) Fattree (b) Mesh

(c) Telstra (d) Ebone (e) Exodus

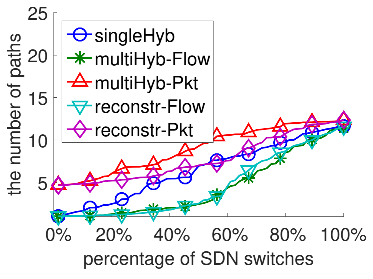
Figure 4.10: Throughput improvement of hybrid forwarding

4.6.2 Hybrid Traffic Engineering

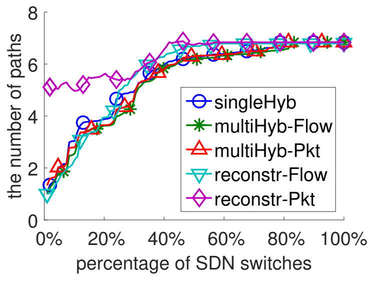
To evaluate the effectiveness of hybrid traffic engineering, we compare the throughput of hybrid forwarding with traditional routing in single path and multipath scenarios. We employ ECMP as flow-level multipath routing in legacy switches, and use equal packet spraying as packet-level multipath forwarding. To show the improvement of hybrid forwarding, throughputs of these hybrid forwarding schemes are normalized by the throughput of traditional single path routing $\frac{T(hybrid,SDN\%)}{T(single,0\%)}$ with different amount of SDN deployment in Figure 4.10. The hybrid forwarding always achieves larger throughput than traditional routing. Meanwhile, the traffic engineering ensures growing throughput with increasing SDN deployment, as the more SDN nodes would introduce more forwarding flexibility. We also find that,



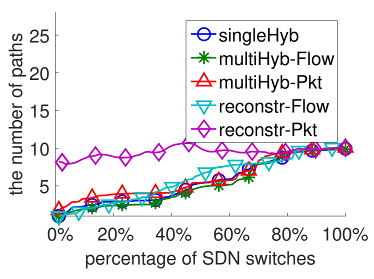
(a) Fattree



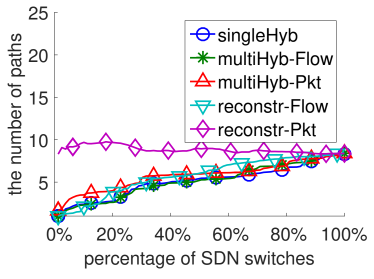
(b) Mesh



(c) Telstra



(d) Ebone



(e) Exodus

Figure 4.11: Path diversity of hybrid forwarding in traffic engineering

with 60% deployment of SDN nodes, the hybrid forwarding with traffic engineering could achieve as much throughput as a full SDN in most experiment networks (except Fat-tree), which means the early upgrade planning and hybrid control are quite important to ensure the performance of the network. In Fat-tree network, as edge switches are upgraded after the core and aggregate switches, this restricts the available path diversity between the edge layer and aggregate layer, and further constrains the forwarding flexibility at the beginning stage of upgrade.

The five topologies in Figure 4.10 show distinct forwarding characteristics. For Telstra, Ebone and Exodus, the hybrid forwarding improves 50% - 100% throughput of traditional routing in both single path and multipath routing scenarios. As the available traditional routing paths for each flow in these network are limited as Figure 4.11 shows, hybrid forwarding enhances the number and diversity of available forwarding paths. However, the hybrid forwarding with multipath routing in Fat-tree

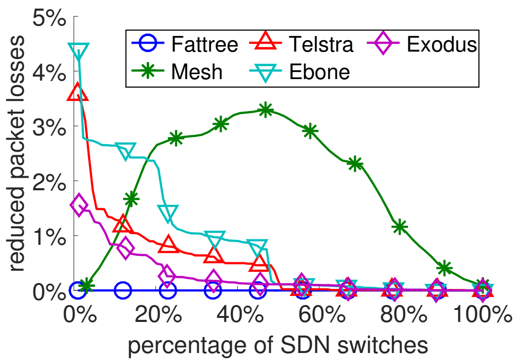


Figure 4.12: Losses reduction for large flows

and Mesh networks gains little improvement, as the traditional multipath routing has already utilized a lot of paths for load balancing in these topologies. Especially with packet-level multipath on legacy switches, packets could utilize more redundant paths in Fat-tree and Mesh networks than ISP topologies even if the SDN deployment is limited, which also results in higher throughput compared to flow-level hybrid multipath forwarding using a single next-hop for each flow on legacy switches.

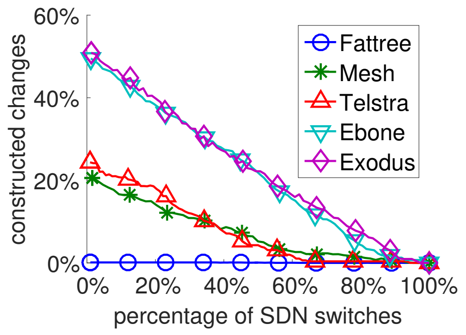
As we consider the forwarding capability of switches in hybrid multipath forwarding, the packet-level hybrid forwarding helps to reduce 0%-5% packet losses compared with straightforward multipath splitting [30] on legacy switches in Figure 4.12. Especially for large flows at the early upgrade stage, legacy switches avoid congesting downstream subpaths with the estimated forwarding capability. Different from the deceasing reduced losses in other networks, Mesh witnesses an increase of reduced losses at the beginning of upgrade and then a decrease with a high fraction of SDN deployment. As flows tend to share links in Mesh compared with the dedicated and localized paths in other topologies, straightforward multipath splitting on legacy switches collides with the forwarding of SDN switches on shared links, and the collision becomes severer with more mixed network devices inside Mesh. When most legacy switches are upgraded to be SDN-enabled, these devices are under the centralized control and avoid straightforward traffic splitting.

4.6.3 Applying Reconstructed Paths

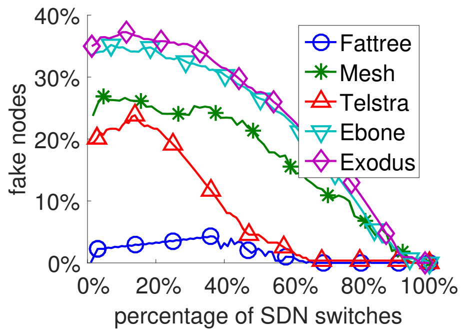
Without the limitation of distributed routing, path reconstruction is expected to achieve more throughput than hybrid forwarding. Figure 4.10 shows path reconstruction could achieve high throughput even if the proportion of SDN deployment is quite small in Telstra, Ebone and Exodus. As Figure 4.11 shows path reconstruction is able to forward a flow with significantly more paths in these ISP topologies, especially for packet-level hybrid multipath routing, so that the throughput experiences remarkable increases in these networks. For Fat-tree and Mesh networks, although there are a lot of available paths, the path reconstruction only reconstructs a similar number of paths compared with hybrid multipath. As links in Mesh are usually shared by a lot of flows, especially in the middle of the network, compulsory multipath may add to congestions on links, so that the path reconstruction selects an appropriate set of paths to ensure the performance.

Although path reconstruction breaks the limitation of traditional routing protocols, the reconstructed forwarding graphs still have to maintain identical next-hops of a legacy switch for different flows to a destination, which restricts the flexibility of path reconstruction. Therefore, path reconstruction also expects more SDN deployment, such that traffic engineering has more freedom to construct a less-cost path flexibly, which is reflected in the growing throughput when SDN deployment increases.

To apply the reconstructed paths, we identify the next-hop changes on legacy switches and inject fake nodes to redirect the traditional routing. Figure 4.13a shows the possibility of next-hop changes on legacy switches in path reconstruction for



(a) Constructed next-hop changes



(b) Fake nodes

Figure 4.13: Next-hop changes and injected fake nodes

each destination compared with traditional packet-level multipath routing. In Fattree, as path reconstruction almost uses the same paths as multipath routing in Figure 4.11a, the possibility of next-hop changes is quite small. For Ebone and Exodus, we construct a lot of new paths which lead to high change possibility in Figure 4.13a. Fake nodes are injected to redirect forwarding of legacy switches in Figure 4.13b. We find that more next-hop changes usually require more fake nodes for redirection. As we never attach a fake node to a SDN node, it ensures a decreasing number of fake nodes when the SDN deployment increases.

4.7 Related Work

Vissicchio et al. [71] suggest that the combination of centralized and distributed paradigms can provide mutual benefits after analyzing a series of hybrid SDN models, which is validated in [70, 89, 90, 91] with various SDN control approaches over legacy hardware. Meanwhile, Vissicchio et al. [92] also show that the coexistence can create forwarding anomalies that ultimately defeat the benefits.

SDN deployment planning: Panopticon [70] investigates the problem of incrementally introducing SDN into an existing network. Hong et al. [74] model SDN deployment and traffic engineering problems to improve network utilization. Caria et al. [93] consider the path diversity in deploying SDN nodes. Poularakis et al. [94] consider the traffic controllability over flows with the reducing upgrade cost during a long upgrade period spanning several years. However, these approaches primarily focus on traffic processing performance or costs in deploying SDN nodes. We further propose the controllability of SDN over the hybrid network which can be independent of the traffic demands.

Traffic engineering in hybrid SDN: A lot of researches utilize the flexibility of SDN to enhance the hybrid network performance. B4 [16] presents the deployment experience of Google's software-defined WAN with various hybrid traffic engineering approaches. Agarwal et al. [30] pioneer the traffic engineering in hybrid SDN with "admissible paths" which are fully controlled by software-defined control, and He et al. [73] complement it with more practical traffic engineering models and solutions. Xu et al. [95] design h-splittable flow routing with "permissible" paths. Wang et al. [96] study hybrid traffic engineering to save energy. We also tried to enhance the effectiveness of hybrid forwarding graphs by designing SDN forwarding in [97]. However, a generalized hybrid traffic engineering approach considering forwarding characteristics of SDN and legacy switches is absent. Our solution is motivated by [30, 73], but we provide a more general model to deal with both the traditional single path and multipath routing protocols.

Limitation of distributed routing: To control the distributed routing conveniently, Fibbing [87] introduces fake nodes, but it mainly focuses on the control over traditional networks and can not be applied in hybrid SDN directly. We adapt Fibbing [87] for hybrid SDN in this chapter. Magneto [98] is a similar approach to Fibbing, but it operates at the data link layer by affecting the forwarding behavior of legacy L2 switches. Guo et al. [99] optimize traffic engineering by adjusting OSPF weights and SDN forwarding. Chu et al. [31] redirect traffic with IP tunnels to avoid link failures. These approaches usually require modifications to distributed routing protocols or configurations on switches, which makes them inconvenient.

4.8 Conclusion

Upgrading a traditional network towards SDN makes the centralized control and traditional distributed network protocols coexist in the hybrid network, and the heterogeneity of the mixed network control complicates the network. To exert the potential flexibility and controllability of SDN in hybrid network during the long upgrade process, in this chapter, we design the SDN placement planning and traffic engineering approaches. Considering the heterogeneous forwarding characteristics of SDN switches and legacy switches, we deploy SDN nodes to maximize the controllability of SDN over the hybrid network. We then consider the forwarding characteristics and capability of switches to maximize network throughput in traffic engineering. Furthermore, to overcome the forwarding limitation of distributed routing, we design a path reconstruction approach and apply the reconstructed paths with fake nodes. The evaluation results with various topologies show that the hybrid traffic engineering ensures better performance especially in the early 60% SDN deployment, and the SDN placement planning guarantees more controllability than existing approaches.

CHAPTER 5
Conclusion

The benefits of flexibility and convenience introduced by SDN make it more and more appealing in both academic and industrial areas, so that SDN maturates over time with the increasing attention. With the decoupling essence of the control plane and data plane in SDN, maintaining a correct, stable and reliable control plane is critical to ensure the network performance. Especially for the correctness, making consistent control decisions efficiently is the key to achieve desired network behaviors. This dissertation presents a series of approaches to ensure effective and consistent control over the network. In this chapter, we first summarize the contributions of this dissertation, and then briefly discuss open issues and future directions of our work, and finally conclude the dissertation.

5.1 Summary Of Contributions

This dissertation enhances the SDN control consistency in three aspects, ranging from flow table update scheduling to the design of control programs. The contributions are summarized as follows:

Efficient data plane update scheduling algorithm: To update flow tables in the data plane consistently and efficiently, we propose an update ordering approach – Cupid. To avoid the heavy overhead of update ordering in previous approaches, we divide the global dependencies among updates into local restrictions by: 1) partitioning a new routing path into several independent segments, 2) identifying critical nodes controlling traffic shifting between the old path and new path, and 3) constructing a dependency graph among critical nodes for potential congested links. We then design a heuristic algorithm to resolve the dependency graph based on the dependency chain to reduce the data plane update completion time. Our simulation shows that Cupid schedules updates at least 2 times faster and has less throughput losses than the state-of-the-art approaches in both fat-tree and mesh networks. Cupid schedules an efficient update order for the decisions made by control programs before applied in the data plane to ensure packets are processed by correct rules.

Control program coordination prototype: To reconcile the SDN control conflicts, we propose a control coordination prototype – Redactor to optimize the consistency and utility of network control in an automatic and dynamic manner. We implement SDN control programs with declarative language Prolog, and compose control programs automatically to execute together to make consistent decisions. To ensure the effectiveness of the generated policy, we use the voting mechanism to select the best policy among the feasible decisions with the most preferences of control programs. When conflicts occur, we use a heuristic approach to compromise a subset of control programs to maximize the control utility. We compare Redactor with the static priority mechanism and Athens [58], and the results show that Redactor always satisfies more control objectives to achieve better control consistency and utility. Redactor integrates control programs flexibly and dynamically to ensure the effectiveness and consistency of generated control decisions.

Design of SDN control in hybrid SDN: To exert the flexibility of SDN control in a hybrid SDN, we propose a solution to handle the heterogeneity caused by distinct forwarding characteristics of SDN control and traditional distributed network routing protocols, therefore boosting the benefits of hybrid SDN. Our solution spans three aspects: 1) planning SDN placement to enhance the SDN controllability over the hybrid network, 2) traffic engineering considering both the forwarding characteristics of SDN and legacy switches, 3) reconstructing and applying optimal forwarding paths to overcome the limitation of distributed routing. The experiments with various topologies show that the SDN placement planning and hybrid forwarding yield better network performance especially in the early 60% SDN deployment, while path reconstruction achieves much higher throughput with more flexibility. This design promises effectiveness and consistency of hybrid SDN control over heterogeneous network devices.

5.2 Open Issues And Future Work

Preserving policies consistency: Cupid only preserves the congestion-free consistency of forwarding paths during data plane updating, and the hybrid SDN control also focuses on the hybrid forwarding correctness and consistency. They do not consider other policies, e.g., waypoint (each packet is required to traverse a certain kind of checkpoint) [100, 101], per-packet consistency [102]. Thus, the application of this dissertation may be limited for other consistency properties. However, preserving all kinds of policies during updating is more complicated than the forwarding consistency. Moreover, enforcing the waypoint policy may even conflict with the loop-free property [101]. While the two-phase commit update [26] could preserve both the correctness and policies, it is highly inefficient with a double number of entries for the initial and final rules on each switch, which is a great challenge for the limited flow table space. Preserving various kinds of policies correctly and efficiently is still an open issue.

Consistent and distributed network knowledge base: In the design of Redactor, we focus on the control conflicts reconciliation by assuming the prototype running upon a common knowledge base. Other newly designed declarative languages for network programming, e.g., [55, 64, 103], also require reliable knowledge bases. Especially for a distributed control plane, synchronization among multiple control nodes is essential to maintain the control plane consistency. However, most distributed control planes offer weak consistency semantics [6]. Although data updates on distinct nodes will eventually be updated on all the control nodes, there is a period of time in which distinct nodes may read different values (old value or new value) for a same property. There are some existing distributed network control platforms supporting the strong consistency (all the control nodes read the most updated property value after a write operation), e.g., Onix [104], ONOS [105]. We can extend these consistent control planes to design distributed knowledge base for declarative queries. However, strict state synchronization may carry excessive performance or complexity penalties [106]. Striking a balance between the consistency and synchronization efficiency requires further exploration.

Consistency in data plane: We only consider the SDN control consistency before applied in the data plane. Actually, there are still consistency issues in the data plane when applying consistent control decisions. Due to the expensive TCAM and limited flow table space [107, 108], mapping all the rules into the data plane may overflow the flow table/TCAM space. Packets failing to match TCAM or flow table entries will trigger requests to software switch or the controller. Although the matching failure does not affect the correctness of network behavior as the control plane has made consistent control decisions, it may add to the processing latency. To reduce frequent requests and long latency, the cached flow entries in TCAM and flow table should have a consistent view with the network traffic considering both temporal and spatial locality. Moreover, since rules with different priorities may overlap in match field, this creates dependencies among the rules in a flow table or TCAM [109, 110]. Simply caching the requested rules without considering the rule dependency may lead to wrong matching and incorrect processing decisions for newly arrived flows [111]. Thus, maintaining consistency of the data plane is also important to apply control decisions correctly. Despite the efforts to maintain consistency on the control plane, a lot of researches [112, 113, 114, 115] are trying to make the data plane more programmable, which complicates the consistency maintenance in data plane.

5.3 Concluding Remarks

While the emerging SDN technology attracts more and more attention and applications with its benefits in flexibility and programmability, controlling the network correctly and efficiently with the new architecture is challenging. This dissertation presents a series of mechanisms to enhance the consistency in making SDN control decisions. We design efficient approaches to reduce the consistency maintenance overhead and avoid the limitations to the control flexibility imposed by the consistency requirements. The contributions in this dissertation provide systematic solutions to ensure an effective SDN control.

There are still a lot of open issues for an absolute consistent SDN, e.g., preserving all kinds of policies, consistent and distributed control plane, consistency in the data plane, etc. A consistent architecture reveals no contradictions when viewed from any perspective [106]. We believe this is a fruitful research area [116], and are excited about future research on designing powerful network control platforms with next-generation programmable switches to make the SDN architecture more effective, stable and reliable.

REFERENCES

[1] Xiaojun Lin, Ness B Shroff, and Rayadurgam Srikant. A tutorial on cross-layer optimization in wireless networks. *IEEE Journal on Selected areas in Communications*, 24(8):1452–1463, 2006.

[2] Jiantao Wang, Lun Li, Steven H Low, and John C Doyle. Cross-layer optimization in tcp/ip networks. *IEEE/ACM Transactions on networking*, 13(3): 582–595, 2005.

[3] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. Intelligent design enables architectural evolution. In *HotNets*, 2011.

[4] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tuxen, Michael Welzl, Dragana Damjanovic, et al. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1): 619–639, 2016.

[5] Saamer Akhshabi and Constantine Dovrolis. The evolution of layered protocol stacks leads to an hourglass-shaped architecture. In *Dynamics On and Of Complex Networks, Volume 2*, pages 55–88. Springer, 2013.

[6] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[7] Andreas Richard Voellmy. *Programmable and Scalable Software-Defined Networking Controllers*. PhD thesis, Yale University, 2014.

[8] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[9] OpenFlow Switch Specification. Version 1.5.1. *Open Networking Foundation*, 2015.

[10] Glen Gibb. *Reconfigurable Hardware for Software-defined Networks*. PhD thesis, Stanford Univerisity, 2013.

[11] Hamid Farhady, HyunYong Lee, and Akihiro Nakao. Software-defined networking: A survey. *Computer Networks*, 81:79–95, 2015.

[12] Software defined networking at scale. URL `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42948.pdf`.

[13] A software defined wan architecture. URL `http://opennetsummit.org/archives/apr12/vahdat-wed-sdnstack.pdf`.

[14] Nader Benmessaoud, Mitch Tulloch, CJ Williams, and Uma Mahesh Mudigonda. *Microsoft System Center-Network Virtualization and Cloud Computing*. Pearson Education, 2014.

[15] Software defined networking (sdn) applied to windows server 2016. URL `https://technet.microsoft.com/en-us/windows-server-docs/networking/sdn/software-defined-networking`.

[16] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[17] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[18] Openflow^{TM}-enabled mobile and wireless networks. *Open Networking Foundation*, 2013.

[19] Yury Andrea Jiménez Agudelo. *Scalability and robustness in software-defined networking (SDN)*. PhD thesis, Universitat Politècnica de Catalunya, 2016.

[20] Weijie Liu. Inter-flow consistency: novel sdn update abstraction for supporting inter-flow constraints. Master's thesis, University of Illinois at Urbana-Champaign, 2015.

[21] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. Survey of consistent network updates. *arXiv preprint arXiv:1609.02305*, 2016.

[22] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[23] Wen Wang, Wenbo He, Jinshu Su, and Yixin Chen. Cupid: Congestion-free consistent data plane update in software defined networks. In *INFOCOM*, 2016.

[24] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *SIGCOMM*, 2014.

[25] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. Scl: Simplifying distributed sdn control planes. In *NSDI*, 2017.

[26] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[27] Framework for SDN: Scope and Requirements. Version 1.0. *Open Networking Foundation*, 2015.

[28] Wen Wang, Wenbo He, and Jinshu Su. Redactor: Reconcile network control with declarative control programs in sdn. In *ICNP*, 2016.

[29] Wen Wang, Wenbo He, and Jinshu Su. Boosting the benefits of hybrid sdn. In *ICDCS*, 2017.

[30] Sankalp Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM*, 2013.

[31] Cing-Yu Chu, Kang Xi, Min Luo, and H. Jonathan Chao. Congestion-aware single link failure recovery in hybrid sdn networks. In *INFOCOM*, 2015.

[32] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[33] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[34] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[35] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network architecture for joint failure recovery and traffic engineering. In *SIGMETRICS*, 2011.

[36] Wen Wang, Wenbo He, and Jinshu Su. M2sdn: Achieving multipath and multihoming in data centers with software defined networking. In *IWQoS*, 2015.

[37] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *HotNets*, 2013.

[38] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *SIGCOMM*, 2013.

[39] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *HotSDN*, 2012.

[40] Rick McGeer. A safe, efficient update protocol for openflow networks. In *HotSDN*, 2012.

[41] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *HotSDN*, 2013.

[42] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. In *INFOCOM*, 2015.

[43] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, 2015.

[44] Lei Shi, Jing Fu, and Xiaoming Fu. Loop-free forwarding table updates with minimal link overflow. In *ICC*, 2009.

[45] Jedidiah McClurg, Hossein Hojjat, and Nate Foster. Efficient synthesis of network updates. In *PLDI*, 2015.

[46] Tal Mizrahi, Efi Saat, and Yoram Moses. Timed consistent network updates in software-defined networks. *IEEE/ACM Transactions on Networking*, 24(6): 3412–3425, 2016.

[47] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: on dependencies, loop freedom, and blackholes. In *IFIP Networking*, 2016.

[48] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[49] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[50] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *SIGCOMM*, 2015.

[51] Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP*, 2009.

[52] Dennis M Volpano, Xin Sun, and Geoffrey G Xie. Towards systematic detection and resolution of network control conflicts. In *HotSDN*, 2014.

[53] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, 2014.

[54] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *NSDI*, 2013.

[55] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *SIGPLAN*, 2011.

[56] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *HotSDN*, 2012.

[57] Andreas Voellmy, Ashish Agarwal, and Paul Hudak. Nettle: Functional reactive programming for openflow networks. In *PADL*, 2011.

[58] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. Democratic resolution of resource conflicts between sdn control programs. In *CoNEXT*, 2014.

[59] Jeffrey C Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. Corybantic: Towards the modular composition of sdn control programs. In *HotNets*, 2013.

[60] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.

[61] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, 2013.

[62] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *NSDI*, 2015.

[63] Mohammad Ashiqur Rahman and Ehab Al-Shaer. A declarative approach for global network security configuration verification and evaluation. In *IM*, 2011.

[64] Timothy L Hinrichs, Natasha S Gude, Martin Casado, John C Mitchell, and Scott Shenker. Practical declarative network management. In *WREN*, 2009.

[65] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[66] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An api for application control of sdns. In *SIGCOMM*, 2013.

[67] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.

[68] Xu Chen, Yun Mao, Z Morley Mao, and Jacobus Van der Merwe. Declarative configuration management for complex and dynamic networks. In *CoNEXT*, 2010.

[69] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *SOSR*, 2016.

[70] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX ATC*, 2014.

[71] Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. Opportunities and research challenges of hybrid software defined networks. *ACM SIGCOMM Computer Communication Review*, 2014.

[72] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Geoffrey G Xie, and Laurent Vanbever. On the co-existence of distributed and centralized routing control-planes. In *INFOCOM*, 2015.

[73] Jun He and Wei Song. Achieving near-optimal traffic engineering in hybrid software defined networks. In *IFIP Networking*, 2015.

[74] David Ke Hong, Yadi Ma, Sujata Banerjee, and Z Morley Mao. Incremental deployment of sdn in hybrid enterprise and isp networks. In *SOSR*, 2016.

[75] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, IETF, 2000.

[76] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[77] Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic engineering with equal-cost-multipath: An algorithmic perspective. In *INFOCOM*, 2014.

[78] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

[79] Dahai Xu, Mung Chiang, and Jennifer Rexford. Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering. *IEEE/ACM Transactions on Networking*, 19(6):1717–1730, 2011.

[80] Mingui Zhang, Bin Liu, and Beichuan Zhang. Multi-commodity flow traffic engineering with hybrid mpls/ospf routing. In *GLOBECOM*, 2009.

[81] Ashwin Sridharan, Roch Guérin, and Christophe Diot. Achieving near-optimal traffic engineering solutions for current ospf/is-is networks. *IEEE/ACM Transactions on Networking*, 13(2):234–247, 2005.

[82] George Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Transactions on Algorithms*, 4(1), 2008.

[83] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *STOC*, 2010.

[84] Lisa K Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4): 505–520, 2000.

[85] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM*, 2000.

[86] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. Mate: Mpls adaptive traffic engineering. In *INFOCOM*, 2001.

[87] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *SIGCOMM*, 2015.

[88] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. In *SIGCOMM*, 2002.

[89] Ryan Hand and Eric Keller. Closedflow: Openflow-like control over proprietary devices. In *HotSDN*, 2014.

[90] Michael Markovitch and Stefan Schmid. Shear: A highly available and flexible network architecture marrying distributed and logically centralized control planes. In *ICNP*, 2015.

[91] Cheng Jin, Cristian Lumezanu, Qiang Xu, Zhi-Li Zhang, and Guofei Jiang. Telekinesis: Controlling legacy switch routing with openflow in hybrid networks. In *SOSR*, 2015.

[92] Stefano Vissicchio, Laurent Vanbever, Luca Cittadini, Geoffrey G Xie, and Olivier Bonaventure. Safe update of hybrid sdn networks. *IEEE/ACM Transactions on Networking*, 2017.

[93] Marcel Caria, Admela Jukan, and Marco Hoffmann. A performance study of network migration to sdn-enabled traffic engineering. In *GLOBECOM*, 2013.

[94] Konstantinos Poularakis, George Iosifidis, Georgios Smaragdakis, and Leandros Tassiulas. One step at a time: Optimizing sdn upgrades in isp networks. In *INFOCOM*, 2017.

[95] Hongli Xu, Xiang-Yang Li, Liusheng Huang, Hou Deng, He Huang, and Haibo Wang. Incremental deployment and throughput maximization routing for a hybrid sdn. *IEEE/ACM Transactions on Networking*, 2017.

[96] Huandong Wang, Yong Li, Depeng Jin, Pan Hui, and Jie Wu. Saving energy in partially deployed software defined networks. *IEEE Transactions on Computers*, 65(5):1578–1592, 2016.

[97] Wen Wang, Wenbo He, and Jinshu Su. Enhancing the effectiveness of traffic engineering in hybrid sdn. In *ICC*, 2017.

[98] Cheng Jin, Cristian Lumezanu, Qiang Xu, Mekky Hesham, Zhi-Li Zhang, and Guofei Jiang. Magneto: Unified fine-grained path control in legacy and openflow hybrid networks. In *SOSR*, 2017.

[99] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, and Jianping Wu. Traffic engineering in sdn/ospf hybrid network. In *ICNP*, 2014.

[100] Stefano Vissicchio and Luca Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *INFOCOM*, 2016.

[101] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *HotNets*, 2014.

[102] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *HotNets*, 2011.

[103] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[104] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[105] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *HotSDN*, 2014.

[106] SDN Architecture. Issue 1.1. *Open Networking Foundation*, 2016.

[107] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.

[108] Jang-Ping Sheu and Yen-Cheng Chuo. Wildcard rules caching and cache replacement algorithms in software-defined networking. *IEEE Transactions on Network and Service Management*, 13(1):19–29, 2016.

[109] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.

[110] Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, Kai Bu, Peng Zheng, Yang Yang, and Chengchen Hu. Ruletris: Minimizing rule update latency for tcam-based sdn switches. In *ICDCS*, 2016.

[111] Bo Yan, Yang Xu, Hongya Xing, Kang Xi, and H Jonathan Chao. Cab: A reactive wildcard rule caching system for software-defined networks. In *HotSDN*, 2014.

[112] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2): 44–51, 2014.

[113] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, dynamic network processing with clickos. In *HotSDN*, 2013.

[114] Wen Wang, Wenbo He, and Jinshu Su. Network intrusion detection and prevention middlebox management in sdn. In *IPCCC*, 2015.

[115] Wen Wang, Cong Liu, and Jun Wang. A more flexible sdn architecture supporting distributed applications. In *COLLABORATECOM*, 2016.

[116] Xin Jin. *Dynamic Control of Software-Defined Networks.* PhD thesis, Princeton University, 2016.