Monte: SFCs Migration Scheme in the Distributed Programmable Data Plane

Xiaoquan Zhang, Lin Cui, *Member, IEEE,* Fung Po Tso, *Senior Member, IEEE*, Yuhui Deng, *Member, IEEE*, Zhetao Li, *Member, IEEE*, and Weijia Jia, *Fellow, IEEE*

Abstract—Service function chains (SFCs) are sequences of network functions that provide specific services to meet operators' needs in today's ISPs and datacenter networks. To improve the performance of SFCs, programmable data planes are used to leverage their low latency and high performance packet processing. However, SFCs need to be adaptable to dynamics such as changes in requirements and attributes. Therefore, the ability to migrate SFCs is essential. Unfortunately, migrating SFCs in distributed programmable data planes is challenging due to the risk of degraded performance and failure to meet SFCs requirements and resource constraints in switches. In this paper, we propose *Monte*, which provides an effective SFCs migration scheme in distributed programmable data planes. We build a novel integer programming model to represent the migration process with constraints on resource limitations of switches and SFCs attributes in the distributed data plane. Additionally, an SFCs migration algorithm is designed to optimize the migration cost by deeply analyzing resource allocation in the switch pipeline. *Monte* has been implemented on both P4 software switches (Bmv2) and hardware switches (Intel Tofino ASIC). Extensive evaluation results show that the migration cost in *Monte* is 94.03% lower on average than the state-of-the-art deployment scheme, and *Monte* can effectively save pipeline resources.

Index Terms—Network Function Migration, P4, Service Function Chain, Programmable Data Plane

1 Introduction

Network Functions Virtualization (NFV) has enabled network functions to be moved from dedicated hardware to software-based modules running on commodity servers. In NFV-enabled networks like ISPs and datacenters, services are implemented as Service Function Chains (SFCs), which consist of multiple network functions chained together in a specific sequence [1]. The emergence of programmable data planes (PDP) based on the protocol-independent switch architecture (PISA) has opened up new possibilities for implementing SFCs, thanks to their flexibility of programmability and high performance [2].

Compared to conventional approaches like middleboxes, the flexible programmability of PDP allows users to easily deploy new functions and update the processing logic directly within the network hardware [3], significantly reducing both deployment and development cost. Moreover, virtualizing network functions on general-purpose servers is often constrained by the CPU's processing power. In contrast, PDP offers line-rate processing capabilities, enabling high-speed data processing (e.g., $\geq$3.2Tbps for Intel Tofino ASIC) with nanosecond level latency, effectively meeting the growing demands of network traffic.

- *Xiaoquan Zhang, Lin Cui, Yuhui Deng, and Zhetao Li are with the National & Local Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Provincial Key Laboratory of Data Security and Privacy Protection, College of Information Science and Technology, Jinan University, Guangzhou, 510632. China. Email: zhangxiaoquan547@gmail.com; tcuilin@jnu.edu.cn; tyhdeng@email.jnu.edu.cn; liztchina@hotmail.com.*
- *Fung Po Tso is with the Department of Computer Science, Loughborough University, UK. Email: p.tso@lboro.ac.uk.*
- *Weijia Jia is with BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai) and BNU-HKBU United International College, Zhuhai, China. Email: jiawj@uic.edu.cn.*

Corresponding author: Lin Cui

Researchers have leveraged the advantageous of PDP to improve the deployment performance of SFCs [4], [5], [6]. However, networks are dynamic, exhibiting either periodic or stochastic patterns of the network functions deployment [7]. To adapt to the changes of service requirements, the updates to SFC usually involve changing the current deployment, including adding new SFCs, modifying network functions in SFCs, and deleting SFCs. Prior deployment works are limited by the assumption of a static network, which hinders them from effectively dealing with dynamics. For example, since states are preserved in distributed switches, even slight changes in demand may result in a large number of unnecessary costs for state migration in deployment schemes. Redeploying updated SFCs in the current network can result in extended downtime. This issue is particularly more critical for programmable switches at the Tbps level of throughput, as the affected network traffic increases significantly during this process. Thus, a live migration scheme is necessary as it is more effective and cost-effective in dealing with dynamic conditions.

However, seamless migration without packet loss in distributed PISA switches is hard to achieve because the two procedures of reconfiguration and state migration will cause downtime. First, *reconfiguration* is to replace the current program with a new program (i.e., replacing logic from old SFCs to new SFCs) in a PISA switch, which will lead to inevitable downtime so as to impact network performance (e.g., 20$\sim$50ms in Intel Tofino switch ASIC). While previous research has explored seamless reconfiguration without downtime in other switching architectures [8], [9], [10], [11], [12], applying these techniques to PISA presents a challenge due to the need for complicated resource optimization and strict sequential updates within its pipeline [8]. Second, *state migration* is to migrate network states from the source switch

to the target switch if network functions with the states need to be migrated. Previous works for distributed PISA switches [13], [14] also present migration time challenges, as they require the entire state migration procedure to be steered in the PDP. However, these existing solutions cannot directly address the problem of dynamic demands and changing network conditions in a distributed PDP.

Migrating SFCs in distributed PDP is a complex task that involves several constraints and objectives. First, migration schemes must respect the constraints of computation and storage resources in the hardware pipeline of PISA switches, such as TCAM, SRAM, and ALUs, as well as SFCs attributes in the substrate network (e.g., throughput and latency). Second, the use of shared logic within network functions to reduce deployment costs further complicates the migration problem. Third, to ensure minimal downtime and interruption to services, a live migration mechanism must be implemented to prioritize migration operations among distributed switches while minimizing migration cost.

In this paper, we present a solution to the challenges of SFCs migration in distributed PDP called *Monte* (Migration in data plane). *Monte* enables the migration of SFCs to adapt to changes of both users' demands and network conditions while considering multiple constraints and objectives. A novel integer programming model is proposed to model migration constraints of resource limitations in the PISA pipeline and SFCs attributes in the substrate network. Moreover, we introduce a migration algorithm that deeply analyzes resource allocation to optimize both migration and deployment costs while adhering to all constraints. To further optimize the live migration process, we propose the migration graph and an associated algorithm that prioritizes the execution sequence of switches in the procedure of reconfiguration and state migration, thereby minimizing migration costs.

In short, the contributions of this paper are three-fold:

1) An SFCs migration scheme on distributed PDP is proposed. We design an integer programming to model the migration problem between resource limitations in PISA and SFCs attributes in the distributed PDP. To the best of our knowledge, this is the first work studying SFCs migration on distributed PDP based on PISA.
2) An algorithm is proposed to optimize the SFCs migration problem by deeply analyzing resource allocation in PISA pipelines among distributed PDP; the migration graph and migration algorithm are proposed to optimize live migration.
3) We have implemented and evaluated *Monte* on both P4 software switches (Bmv2) and hardware switches (Intel Tofino ASIC) to demonstrate its effectiveness: migration cost in *Monte* is 94.03% lower than the previous deployment scheme on average while *Monte* can effectively save pipeline resources.

The rest of the paper is organized as follows. Section 2 gives backgrounds and related works. Section 3 introduces problem formulation. Section 4 details algorithm design. Section 5 discusses the implementation and Section 6 shows the experiment results. Section 7 concludes the paper.

2 Backgrounds and Related Works

2.1 Constraints of the pipeline in PISA

PISA is widely adopted in P4 hardware switches. Users can implement P4 programs to define how to process network packets in the pipeline. The pipeline architecture consists of dozens of stages, each of which contains equivalent compute and memory resources that cannot be reassigned to other stages. As shown in Figure 1, each stage has SRAM and TCAM resources for storage and ALUs (gray trapezoids) for computation (other resources are not listed due to the space limit). Some essential programmable components in P4 language can be achieved by handling these resources. The match-action table is used for mapping keys in packet headers to actions, whose entries can be saved by SRAM or TCAM if they apply an exact or ternary match. Registers are commonly used to save network states in sketch applications, which mainly consume SRAM resources to preserve network states and ALUs to operate network states (e.g., addition). Additionally, two match-action tables must be placed in different stages if they have a dependency relationship (details in Section 6).

SFCs deployment in the pipeline must respect the above resource and computation constraints per stage. For example, in Fig. 1, the table *Sketch2* (purple) of *NF2* must be placed in two stages since it needs memory resources of two stages, and it must be placed after the table *Hash* (green) because they have dependency relationship (i.e., the output of *Hash* is the input of *Sketch2*).

2.2 Reconfiguration and state migration in PISA

The migration process in PISA consists of two non-trivial procedures: **reconfiguration** and **state migration**. First, Reconfiguration is the process of configuring a new P4 program to replace the old one when SFCs need to change. This incurs downtime, causing packet loss and service interruption. Making runtime reconfiguration is challenging due to tight coupling in the PISA pipeline [8]. For example, removing or adding match-action tables in a stage may incur resource reallocation or fragmentation, making resource optimization complex. Moreover, these table operations cannot execute in parallel, prolonging the downtime of reconfiguration. Previous works discussed runtime reconfiguration in other programmable data planes [8], [9], [10], but these solutions either were not pipeline architectures [8], [10] or proposed new packet switching architecture [9]. They cannot be directly applied to PISA switches.

Second, P4 allows applications to reference per-flow states by hashing specific packet headers. Simply migrating these per-flow states by a centralized control plane may not work [13]. This is because the location of per-flow states is device-specific. Different switches might apply different hash operations (e.g., hash algorithm and inputs) or maintain different sizes for registers. Migrating states totally in the PDP can avoid the problem but produce extra migration time. Swing state [13] piggybacks network states on network traffic to steer state migration in PDP without the involvement of the control plane. P4Sync [14] leveraged the packet generator to proactively generate packets that piggyback all states at a tunable speed, which accelerates migration speed

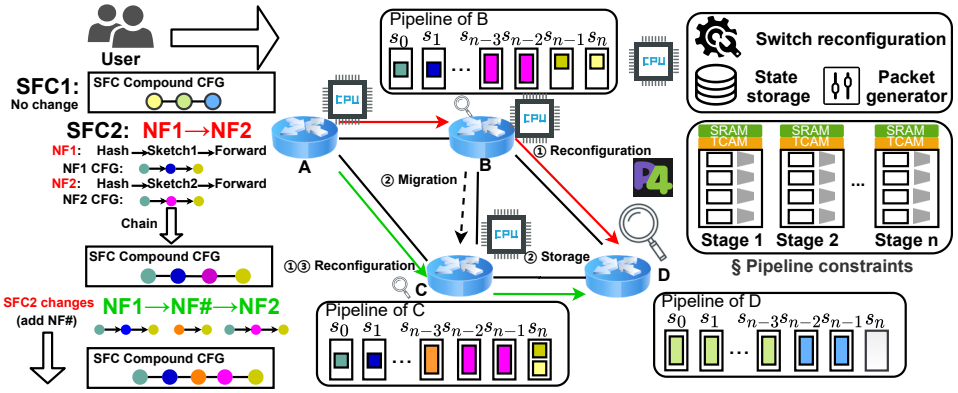


Fig. 1. Overview of SFCs migration in *Monte*. There are two SFCs deployed in the switch B and D. *Monte* receives SFC changes for *SFC2*, i.e., adding a new *NF#* and migrates it from switch B to C due to requirements of performance (e.g., latency and throughput). After migration completion, SFC deployment in the switch C respects constraints in the pipeline and SFC performance attributes in the substrate network.

and guarantees the completeness of state migration. However, how to minimize migration time among distributed programmable switches still remains unexplored.

2.3 SFCs deployment in programmable data planes

Extensive research has explored the deployment of SFCs in PDP. The SFCs deployment is to steer network traffic traversing specific fixed network functions in order among multiple switches [4], [5], [15]. P4NFV [5] and FlexMesh [4] provided network architectures to dynamically implement SFCs for satisfying network requirements on runtime. SFP [15] proposed a mathematical programming model to maximize traffic processing. Some works have discussed offloading partial logic for processing acceleration [16], [17], [18]. DPPx [16] leveraged a centralized controller to offload network functions and manipulate the deployment of SFCs and traffic route. P4SFC [17] and LightNF [18] emphasized partially offloading logic which can be accomplished in the P4 data plane. Utilizing recirculation operations can achieve SFCs deployment in a single switch [19], [20]. Dejavu [20] implements SFCs by forcing packets to enter the processing pipeline multiple times. However, throughput sharply decreases by 95% when the number of recirculation reaches 5. In order to reduce deployment cost, some works focus on removing the same logic within different network functions to save limited resources in the pipeline [6], [21]. pSFC [6] proposed an ILP that allows composing the same logic and offered a scheme to embed SFCs onto the distributed P4 data plane. For example, in Fig. 1, *NF1* and *NF2* in the SFC have the same logic (*Hash* and *Forward* tables), and they can be composed to be placed in the same stage s_{n-1} in switch B.

However, these deployment schemes are based on the assumption that network demand is deterministic and resources are not oversubscribed. They are ineffective for dealing with network dynamics because any tiny changes in SFCs could lead to a large number of unnecessary migrations with large migration cost.

3 System Model

3.1 Overview of *Monte*

The system architecture of *Monte* is shown in Fig. 1. Upon receiving SFCs changes from operators, *Monte* analyzes each SFC by the graph tool (Section 3.3), and models constraints in the hardware pipeline (Section 3.4) and the substrate network (Section 3.5). Then it computes a dynamic migration scheme (Section 4.1) with minimal migration cost (Section 4.2).

To achieve the two procedures of reconfiguration and state migration, the CPU in P4 hardware switches is responsible to operate three modules: switch reconfiguration, state

TABLE 1
Notation of symbols

Control flow graph (CFG)	
G_c	Control flow graph of SFCs, $G_c = \{V_c, E_c\}$, including node set $n_i, n_j \in V_c$ and edge set E_c
V_t	A set of type $t \in T$ of nodes
V_f	A set of SFC f of nodes
$D(n_i, n_j)$	Dependency between node n_i and n_j
I_i, O_i	Input and output stage for a node n_i
f_{start}, f_{end}	Input and output switches of SFC f
R_i^m	Resource requirement of type m of node i
U^m	Limited type m of resources in each stage
Substrate network	
G_d	Substrate network graph, $G_d = \{V_d, E_d\}$, including node set $u, v \in V_d$ and edge set $uv \in E_d$
S_u	A set composed of stages of programmable switch u.
$l_{uv}, \varphi_{delay}^f$	Latency of link uv and end-to-end latency of SFC f
ζ_{bw}^f, b_{uv}	Bandwidth of SFC f and link uv
ρ_u	Whether switch u is programmable
h	Maximum number of available substrate switches
Migration graph	
G_r	Migration graph, $G_r = \{V_r, E_r\}$, including node set V_r and edge set E_r
τ	Migration scheme, including reconfiguring switch set τ_{re} and state migrating switch set τ_{st}
C_{st}^u, C_{re}^u	Cost of stateful migration and switch reconfiguration of substrate switch u
C_{dc}	Cost of deployment
Variables	
x_i^s	Whether CFG node n_i is placed in stage s
x_t^s	Whether stage s places type t of CFG node(s)
$Q_i^{s,m}$	Allocated resources of type m for node n_i in stage s
$Q_t^{s,m}$	Allocated resources of type m for type t of node(s) in stage s
z_s^f	Whether SFC f has CFG node(s) placed in stage s
z_u^f	Whether SFC f has CFG node(s) placed in switch u
z_{uv}^f	Whether the SFC f traverses the substrate link uv

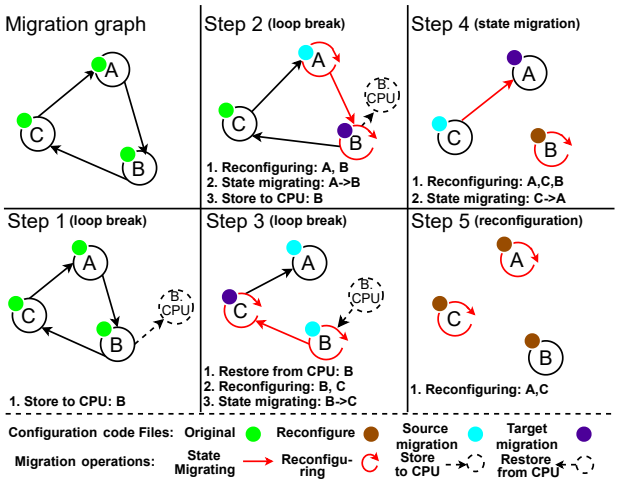


Fig. 2. An example of migration graph.

storage, and packet generator. A simple example of state migration shows how the three modules work in Fig. 1. In the migration procedure, in step 1, the module of switch reconfiguration is responsible for reconfiguring switches B and C. Switch B that is reconfigured from the original P4 program to the source migration program piggybacks states of *NF2* saving in registers to packets, and switch C that is configured to the target migration program receives piggybacking packets and copies them in the CPU for storage. In step 2, the CPU of switch B generates piggybacking packets by the packet generator and sends them to switch C at an adjustable speed. In step 3, once all states are migrated successfully, the switch C is configured to the new P4 program (adding the new *NF#*) while its CPU recovers migrated states which reside in registers of the pipeline.

3.2 Live migration mechanism

A live migration mechanism is crucial to ensure the proper sequence of migration operations applied in corresponding switches, thereby avoiding migration conflicts. For instance, if state migration is required from switch A to B and B to C, a conflict arises in switch B if the migrations are executed in parallel. To address this issue, we propose the *migration graph* that illustrates the migration request. The graph is a directed graph $G_r = \{V_r, E_r\}$, where V_r is the set of nodes representing the migrated switches, and E_r is the set of edges representing the state migration request from the source node to the target node. The live migration mechanism is responsible for applying all requests of state migration and switch reconfiguration in the correct order, as specified by the migration graph. Therefore, *Monte* transforms the problem to the procedure of finding the topological sequence of the migration graph. For example, applying state migration can remove edges, and nodes with 0 in-degrees can be added to the sequence result.

To process the migration graph, we define four operations: (1) *state migrating*, which transfers states from the source switch to the target switch; (2) *store to CPU*, which retrieves states from hardware pipeline and saves them to the CPU; (3) *reconfiguring*, which reconfigures hardware pipeline from the original file to the reconfigure file; (4) *restore from CPU*, which restores states from CPU to hardware pipeline. Multiple switches can execute operations in parallel to reduce migration time if they do not conflict. In addition, four P4 code files are maintained in each substrate switch. The original file is used to configure the current pipeline, while the reconfigure file is used to configure the new pipeline. The source migration file is used to migrate states in the current pipeline to others, while the target migration file is used to receive migrated states. For instance, the reconfiguring operation is to replace the original file with the reconfigure file; if switch A migrates states to B, A and B should configure the source migration file and the target migration file respectively for state transmission.

Fig. 2 illustrates an example of a migration graph that demonstrates the migration process via the four operations. In this example, three substrate switches are source and target migrated switches simultaneously. The migration graph is not a loop-free graph, which means that there may be cycles in the graph, such as $A \rightarrow B \rightarrow C \rightarrow A$. To break the loop, a node needs to be chosen first (e.g., B) to reserve its states into a backup node. An effective method to achieve this is to use the CPU in the P4 hardware switch as the backup. The advantage of using the CPU is that it does not incur any migration cost, and the process can easily be achieved by PCIe tunnel [22]. In step 1, the CPU of switch B retrieves and saves states that need to be migrated to C. In step 2, switches A and B execute reconfiguring operations to replace with source and target migration files, respectively, and then switch A migrates its states to B. After state migration process is completed, the CPU of switch B retrieves the migrated states. In step 3, switch C is reconfigured to receive migrated states, and switch B restores the states of the original pipeline from its CPU. Then, switch B migrates states to C while CPU operations in C are the same as those in step 1 and 2. In step 4, operations are applied for the state migration from switch C to A, where switch B can configure the reconfigured file in parallel. Lastly, switch A and C are reconfigured to the reconfigure files. A migration graph can be illustrated by three procedures, loop break, state migration and reconfiguration. Each procedure can further be achieved by the four operations in a proper sequence.

3.3 Network function abstraction

Different graph models can be used to abstract network functions for analyzing elements of P4 programs (e.g., match-action tables) [6], [23]. The CFG (Control Flow Graph) [6] details more properties of P4 programs. It is defined as a directed acyclic graph $G_c = \{V_c, E_c\}$, where V_c is the set of nodes that refers to tables or conditions in P4 programs, and E_c is the set of directed edges each of which describes the execution direction of two nodes. Each node $n_i \in V_c$ has two attributes: (1) required resources and (2) fields of match and action. The first attribute describes required resources to allocate when the node is placed in the stage (e.g., SRAM, TCAM, and ALUs). The second attribute dictates what packet fields are matched and applied to actions. For example, the node of *forward* table in P4 program has a match field of the destination IPv4 address which matches each packet's IPv4 to corresponding table entries.

In the hardware pipeline, if two nodes mutually operate the same fields, they have the dependency relationship

and cannot be placed in the same stage. For example, if node n_i modifies the same fields node n_j matches, n_i must precede n_j to avoid inconsistent problems in the pipeline. Other dependency relationships are detailed in Section 5. A dependency matrix D is defined to represent relationships among different nodes in the CFG, where $D(n_i, n_j) = 1$ if the two nodes have a dependency.

An SFC can be modeled by a compound CFG formed by several CFGs of chained network functions, where the same logics within these CFGs can be merged to share allocated resources in the P4 pipeline [6]. As shown in Fig. 1, the compound CFG of the SFC is composed of two CFGs of *NF1* with three nodes and *NF2* with four nodes, and the yellow node is the same node (e.g., *forward* table) in both CFGs.

3.4 Constraints in the hardware pipeline

The placement of P4 programs in the hardware pipeline can be transformed as embedding nodes of a CFG while satisfying resource constraints and correct dependencies among nodes. Multiple SFC requests from users can be analyzed and further merged into a single CFG consisting of CFG nodes $n_i \in V_c$. We use two node sets, V_f and V_t, to represent two sets of nodes. Nodes in V_f belong to SFC $f \in F$. Nodes with the same properties (i.e., shared nodes) exist within the CFGs. Assume that the node set has T of node types (e.g., *Forward*, *Sketch1*, *Sketch2*, and *Hash* in Fig. 1), then we use V_t to represent the set of type t nodes ($t \in T$). If $n_i, n_j \in V_t$, n_i and n_j are shared nodes. Two binary variables, x_i^s and x_t^s, are used to denote whether the node n_i is placed in stage s and whether the s places type t node(s), respectively. Thus, the relationship between x_i^s and x_t^s is therefore:

$$\forall t \in T, \forall s \in S : x_t^s = \bigvee_{n_i \in V_t} x_i^s, \tag{1}$$

where S denotes the set of available stages of all programmable switches in the substrate network topology.

To completely deploy SFCs, each node n_i in the node set must be assigned to at least one stage in the pipeline:

$$\forall n_i \in V_c : 1 \leq \sum_{s \in S} x_i^s \leq \xi, \tag{2}$$

where ξ is a constant to confine the maximum number of stages within which a node can be placed.

I_i and O_i are defined as the input and output stages of n_i. If two nodes have a dependency (i.e., $D(n_i, n_j) = 1$), the output stage of node n_i must precede the input stage of n_j.

$$\forall n_i, n_j \in V_c : D(n_i, n_j) = 1 \implies O_i < I_j. \tag{3}$$

Meanwhile, a node may consume different types of pipeline resources $M = \{SRAM, TCAM, ALU\}$ for computation and storage. An integer variable $Q_i^{s,m}$ represents the allocated resources of type $m \in M$ for node n_i in stage s. All resources of each node must be allocated completely:

$$\forall n_i \in V_c, \forall m \in M : \sum_{s \in S} Q_i^{s,m} = R_i^m, \tag{4}$$

where R_i^m is the requirement for resource type m of n_i.

Two shared nodes, say $n_i, n_j \in V_t$, should be allocated with the same amount of resources in a stage:

$$\begin{aligned} \forall s \in S, \forall n_i, n_j \in V_t, \forall t \in T, \forall m \in M : \\ x_i^s = x_j^s \implies Q_i^{s,m} = Q_j^{s,m} \end{aligned} \tag{5}$$

Since each stage in the hardware has limited resources U^m, the total allocation of resources of nodes in a stage must not exceed its capacity:

$$\forall s \in S, \forall m \in M : \sum_{t \in T} Q_t^{s,m} \cdot x_t^s \leq U^m, \tag{6}$$

where $Q_t^{s,m}$ refers to allocated resources for type t of node(s). Under the Equation (4)$\sim$(6), resources in the pipeline are allocated according to the type of nodes. In other words, deploying multiple nodes of the same type in the same stage does not result in duplicated resource consumption, thereby allowing the shared logic to reduce deployment costs. Specifically, we use $Q_t^{s,m}$ to accumulate the resource consumption in stage s instead of $Q_i^{s,m}$. When multiple nodes of the same type t are deployed in the same stage (i.e., $x_t^s = 1$), the resource consumption is accounted for only once using $Q_i^{s,m}$ rather than multiple times using $Q_i^{s,m}$ (Equation (6)).

3.5 Constraints in the distributed substrate network

The substrate network can be abstracted as a graph $G_d = \{V_d, E_d\}$, where V_d represents the set of switches (including programmable switches and non-programmable forwarding switches) and E_d represents the set of links. The stage set of switch $u \in V_d$ in the physical network is defined as S_u. Two binary variables are used to locate which switches are placed nodes of SFCs. z_s^f refers to whether stage s places node(s) belonging to SFC $f \in F$, and z_u^f represents whether the SFC f is deployed in the switch $u \in V_d$. Locations of SFCs can therefore be deduced:

$$\forall s \in S, \forall f \in F : z_s^f = \bigvee_{n_i \in V_f} x_i^s \tag{7}$$

$$\forall u \in V_d, \forall f \in F : z_u^f = \bigvee_{s \in S_u} z_s^f \cdot \rho_u, \tag{8}$$

where $\rho_u = 1$ represents that u is a programmable switch that can be assigned to deploy SFCs; otherwise, the switch is a non-programmable forwarding switch ($\rho_u = 0$).

Next, a binary variable z_{uv}^f is introduced to represent whether an SFC f traverses the substrate link uv. To guarantee only one single path for the deployment of an SFC f, the ingress and egress switches must connect and be connected to only one switch:

$$u = f_{start} : \sum_{v \in V_d} z_{uv}^f = 1; u = f_{end} : \sum_{v \in V_d} z_{vu}^f = 1. \tag{9}$$

A switch in the middle of the path must only connect or be connected to one switch:

$$\forall u \in V_d \setminus \{f_{start}, f_{end}\} : \sum_{v \in V_d} z_{uv}^f \leq 1, \sum_{v \in V_d} z_{vu}^f \leq 1. \tag{10}$$

Afterward, the Equation (11) can guarantee that packets can be steered to a path from ingress to egress of the SFC in the substrate network.

$$\sum_{v \in V_d} \sum_{f \in F} (z_{uv}^f - z_{vu}^f) = \begin{cases} 1, & u = f_{start} \\ -1, & u = f_{end} \\ 0, & \text{otherwise} \end{cases}, \tag{11}$$

where f_{start} and f_{end} denote the ingress and egress switches of f, respectively.

Switches that are allocated resources for an SFC must be located in the path of the SFC:

$$\forall u \in V_d \setminus \{f_{end}\} : z_u^f = 1 \implies \sum_{uv \in E_d} z_{uv}^f \geq 1. \quad (12)$$

The latency of an SFC is restricted to the maximum allowed end-to-end latency φ_{delay}^f:

$$\forall f \in F : \sum_{uv \in E_d} l_{uv} \cdot z_{uv}^f \leq \varphi_{delay}^f, \quad (13)$$

where l_{uv} represents latency of link uv.

The link bandwidth capacity b_{uv} cannot be exceeded:

$$\forall uv \in E_d : \sum_{f \in F} \zeta_{bw}^f \cdot z_{uv}^f \leq b_{uv}, \quad (14)$$

where ζ_{bw}^f represents the bandwidth of SFC f.

3.6 Migration cost and objective

The migration process involves modeling the migration scheme ($\tau = \tau_{re}, \tau_{st}$) from the given deployment (e.g., variables) to the migrated deployment. The migration scheme includes switches that execute the sorting sequence of two migration operations (reconfiguring switch set τ_{re} and state migrating switch set τ_{st}). The cost of the migration operation is determined by the amount of network traffic packets that are affected during the execution procedure. To calculate the migration cost, we define the cost of reconfiguring C_{re}^u and state migrating C_{st}^u in switch u as the downtime (explained in Section 6) of switch applying operations of reconfiguring and state migrating multiplied by the flow rate of traffic flows in switch u. Thus, the migration cost can be obtained as the sum of all migration operations in the live migration.

The migration objective aims to minimize the migration cost and deployment cost C_{dc} which is defined as the number of occupying stages [6]. To simplify the problem, we use the weighted sum method to transform it into a single objective problem. The objective therefore can be defined as:

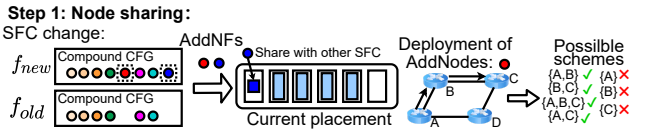
$$\min(\omega \cdot (\sum_{u \in \tau_{re}} \cdot C_{re}^u + \sum_{v \in \tau_{st}} \cdot C_{st}^v) + (1 - \omega) \cdot C_{dc}). \quad (15)$$

The problem of P4 program placement across multiple switches has been shown to be NP-complete [6], [24]. Specifically, by removing the dependencies between nodes in the CFG, the problem reduces to a multi-knapsack problem (MKP). In *Monte*, when the SFC requirements change, the deployment strategy must still comply with the pipeline constraints imposed by the switches. Given that the same hardware switches are utilized, these constraints remain unchanged. Consequently, this problem is also NP-hard.

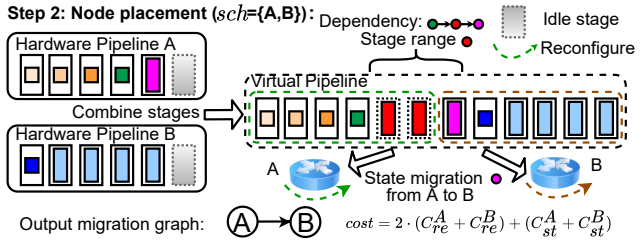
4 Algorithms of *Monte*

4.1 SFCs migration algorithm

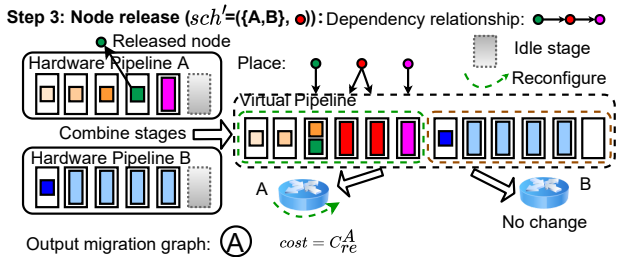
A resource-efficient SFCs migration algorithm is proposed to search for the migration scheme with minimal cost (Algorithm 1). A migration scheme consists of migrated switches that execute migration operations and other switches in the substrate network remain unchanged. Given such a scheme, the algorithm attempts to place new nodes of NFs from new SFCs requests under constraints in the hardware pipeline and substrate network, and finally computes migration cost. The main idea of the algorithm can be summarized into three steps (an example shown in Fig. 3).



(a) In step 1, the algorithm compares the differences of an SFC before and after changes (i.e., the red and dark blue nodes are added). Since the dark blue node has already been deployed and is used by other SFCs, it can be directly shared without allocating additional resources. The algorithm calculates possible schemes based on the current deployment status to place the red node, where a single switch is estimated to have insufficient resources to place the red node.



(b) In step 2, with the scheme $\{A, B\}$, the resources of the two switches are combined into a virtual pipeline. The deployment of the red node must consider the stage range determined by its dependencies with other nodes. Consequently, the red node is deployed in A, and the state of the purple node needs to be migrated to B.



(c) In step 3, the red node must be placed after the green node due to dependency constraints. By shifting the placement of the green node to an earlier stage (i.e., releasing node dependency), sufficient resources become available in A to accommodate the red node. As a result, the migration scheme does not require any changes to the deployment in B, thereby reducing the migration cost.

Fig. 3. An example to illustrate the migration algorithm.

Step 1: Node sharing. The algorithm first compares differences between the old and new request of each SFC (line 4 $\sim$ 5 in Algorithm 1, $f_{new} \in F_{new}$), resulting in *addNFs* that saves nodes of new network functions and deleted nodes from f_{old} that are added in *DelNodes*. Since placing shared nodes in the same stage can avoid resource wastage (Equation (4)$\sim$(6) in Section 3.4) and does not cause reconfiguration (detailed in Section 5), the algorithm then checks each node in *addNFs* whether they have shared nodes that are placed in the pipeline for other SFCs (line 6 $\sim$ 11). If so, the node would be placed in the same stage; otherwise, the node is added in *AddNodes*. In addition, since a node may have dependency with other nodes (Equation (3)), there is a valid stage range within which the node can be placed,

and the same stage (shared nodes are placed in) must be in the range (line 7 $\sim$ 8 of Algorithm 1).

A set of migration schemes (*Sch*) is generated (line 13), The algorithm prioritizes generating migration schemes that consist of all possible combinations of switches where SFCs are deployed. For example, in Fig. 3(a), since a single switch is estimated to lack sufficient resources to place the red node, four remaining possible schemes are considered. If these migration schemes do not meet the constraints of the SFCs in the substrate network (Section 3.5), Breadth-First Search (BFS) is used to search for new switches where the SFCs have not yet been deployed until the SFC constraints are met [25]. The search results are then used as the migration schemes.

It is noted that the delete process (line 12) of *DelNodes* can avoid cost happening in the migration (details in Section 5).

Step 2: Node placement. Nodes in *AddNodes* are further placed by function *Node_place()* (Algorithm 2). Sorting to determine which node can be placed earlier is important as it can affect the quality of placement [23]. To evaluate each node, *Monte* selects three critical sorting $Metrics$: memory consumption, stage range, and the number of dependencies with other nodes, because their values are the main factors that may lead to placement failure. For example, a small stage range narrows the range of available stages in that the node can be placed, and large memory consumption means that more available resources are needed. We use min-max normalization to rescale each metric in the range $[0, 1]$:

$$\forall k \in Metrics, \forall n_i \in V_c : \kappa_i^k = \frac{W_i^k - W_{min}^k}{W_{max}^k - W_{min}^k}, \quad (16)$$

where W_i^k and κ_i^k denote the value and rescaled value of metric k of node n_i, W_{max}^k and W_{min}^k refer to the maximum and minimum value of metric k.

The total of the three metrics ε_i of node n_i is used to sort the nodes from large to small (line 2 of Algorithm 2):

$$\forall n_i \in V_c : \varepsilon_i = \sum_{k \in Metrics} \kappa_i^k. \quad (17)$$

Node placement in multiple hardware pipelines is complex as it must consider all constraints (Section 3.4) among these pipelines. Thus, to optimize resource allocation, *Monte* combines all stages in a virtual pipeline (line 5 of Algorithm 2). For example, in Fig. 3(b), idle stages (without placing nodes) from different pipelines can be flexibly moved and combined to satisfy the stage range of the red node. Specifically, the two idle stages (grey rectangle) are moved to before the purple node for the placement of the red node.

However, state migration may happen in recovering the virtual pipeline to the hardware pipelines. As in Fig. 3(b), the purple node is moved to switch B so that state migration of its states from A to B is required. Fortunately, if *Monte* rules that all original relative positions of stages are static when moving idle stages, a migration loop will not happen, because loop migration must incur disorder of relative positions between two stages.

Afterward, each node in *AddNodes* is placed into the virtual pipeline in order while updating its migration graph via function *Place()* until it is placed successfully or no idle stage can be used to place. In the placement procedure, we define a high-cost node as a node that leads to state migration or placement failure, increasing the migration cost. Whenever a high-cost node is detected, the node and the current scheme are recorded for further analysis (line 9 of Algorithm 2). Once all nodes in a scheme are placed successfully, the algorithm computes the migration cost using the *Compute_cost* function (Section 4.2). It outputs the best scheme with minimum migration cost and a set of schemes that need to be placed again, denoted as *Sch*$'$.

Step 3: Node release. This step is to deeply analyze high-cost nodes and their schemes. In the first place, function *Node_release()* releases nodes (i.e., $n_i \in F_{old} \setminus AddNodes$) that have dependency with the high-cost node from the given deployment. These nodes will be attached to *Sch*$'$ and placed again. The strategy works well because it releases constraints of dependency of high-cost nodes, thus enlarging the stage range of the node so as to increase the possibility of successful placement. In Fig. 3(c), the green node has dependency with the red node. After releasing the green node and placing it again, the red node can be fully placed in switch A, eliminating the need to change the deployment of switch B, thereby decreasing migration cost.

Next, function *Node_place'()* places the previous results. Different from function *Node_place()*, this function uses BFS to search new substrate switches for placement if the virtual pipeline has no available stage resources. The function provides sufficient stage resources until available substrate switches are used up, but it also brings more deployment cost. Therefore, *Monte* provides a tuning parameter h to limit the number of new extra substrate switches that are used to place. For example, if h equals 0 no new substrate switch can be used. More experiments about the parameter are in Section 6.2. The function outputs the best scheme from *Sch*$'$.

Lastly, the algorithm selects the minimal cost scheme between the two migration schemes and executes SFCs migration based on the sorting sequence of migration operations in the network. The complexity of Algorithm 1 is determined by function *Node_place()*. Since the function requires trying all possible schemes and, for each scheme, each node in *AddNodes* should search placing stages from its stage range, its complexity is $\mathcal{O}(\mathbb{M} \cdot \sigma \cdot \mathbb{R})$, where $\mathbb{M}$ is the number of all possible schemes, σ is the number of nodes in *AddNodes*, and $\mathbb{R}$ is the length of the stage range of nodes.

4.2 Live migration procedure

Algorithm 3 is proposed to illustrate live migration and compute the cost of a migration graph (as defined in Section 3.6). It first searches whether loops exist in the migration graph and breaks them (line of 1 $\sim$ 8). In each iteration, the algorithm handles one loop, in which the node that has the maximum in-degrees would be selected to break because it can reduce the complexity of the graph. Then, it adds the node in V_{rel} and updates the migration graph. Once all loops are broken, function *Level_sort()* is used to sort nodes in levels based on the thought of topological sorting (line 9), where nodes with zero in-degrees are classified in the same level. Besides, since state migrating operations need to identify target migration switches, results of V_{rel} and V_{sort} also save target switches for each state migration request. Afterward, switches execute reconfiguring operations and state migrating operations can be classified as τ_{re} and τ_{st} from the two nodes set.

Algorithm 1: SFCs migration algorithm

Input: $\mathbb{D}$, the given deployment
F_{new}, F_{old}, migrated SFCs and current SFCs in $\mathbb{D}$
Initialize *AddNodes*, *DelNodes*
if F_{new} != F_{old} **then**
  for *each* f_{old} *in* F_{old} **do**
    addNFs$\leftarrow f_{new} - (f_{old} \cap f_{new})$
    DelNodes $\leftarrow$ *DelNodes* $\cup (f_{old} - (f_{old} \cap f_{new}))$
    for *each* n_i *in addNFs* **do**
      range=*Compute_stage_range*(n_i)
      if $\forall t \in T, n_i, n_j \in N_t : x_j^s = 1, s \in$ *range* **then**
        n_i is placed in the stage s
      else
        AddNodes.Insert(n_i)
  Delete_nodes(DelNodes)
  Sch $\leftarrow$ *Generate_schemes*(*AddNodes*, $\mathbb{D}$)
  C, V_{migra}, *Sch*$' \leftarrow$***Node_place***(*Sch*,*AddNodes*, $\mathbb{D}$)
  Sch$' \leftarrow$ *Node_release*(*Sch*$'$, $\mathbb{D}$)
  $C', V'_{migra} \leftarrow$*Node_place*$'$(*Sch*$'$, h, $\mathbb{D}$)
  if $C < C'$ **then**
    Migration(V_{migra})
  else
    Migration(V'_{migra})

Algorithm 2: Function: ***Node_place()***

Input: $\mathbb{D}$, the given deployment
AddNodes, the set of nodes that need to be placed
Sch, possible migration schemes
Initialize $C \leftarrow$*inf*, V_{migra}, *Sch*$' \leftarrow \emptyset$
AddNodes.Sort_by_metrics()
for *each sch in Sch* **do**
  $G_r^{sch} \leftarrow \emptyset$
  Combine_stages(sch)
  for *each* n_i *in AddNodes* **do**
    $flag, G_r^{sch} \leftarrow$*Place*($n_i, \mathbb{D}, sch, G_r^{sch}$)
    if $!flag$ **then**
      Sch$'$*.Insert*($\langle sch, n_i \rangle$)
  if *Place successfully* **then**
    cost, $V_{mig} \leftarrow$ ***Compute_cost***(G_r^{sch})
    if *cost*$< C$ **then**
      $C \leftarrow$*cost*
      $V_{migra} \leftarrow V_{mig}$
Return C, V_{migra}, *Sch*$'$

The function finally outputs the two migration sorting sequences, presenting the migration process. Specifically, in V_{sort}, nodes with 0 in- and out-degrees execute reconfiguring operations, nodes with 0 in-degrees and 1 out-degree apply state migration to their target switches, and nodes in V_{rel} execute the procedure of loop break. Therefore, the migration cost can be computed as line 10 of Algorithm 3.

The complexity of Algorithm 3 is $\mathcal{O}(\psi \cdot (|V_r| + |E_r|))$,

Algorithm 3: *Function:* ***Compute_cost()***

Input: G_r, migration graph
while *True* **do**
  Loopnodes, $flag \leftarrow G_r$*.Find_loop()*
  if $flag$ **then**
    $node \leftarrow \arg\max_{u \in Loopnodes} Indeg(u)$
    V_{rel}*.push(node)*
    G_r*.Update_graph(node)*
  else
    break
$V_{sort} \leftarrow G_r$*.Level_sort()*
cost $\leftarrow \sum_{u \in \tau_{re}} C_{re}^u + \sum_{v \in \tau_{st}} C_{st}^v, \tau_{re} \cup \tau_{st} = V_{rel} \cup V_{sort}$
return *cost*, $V_{sort} \cup V_{rel}$

where ψ represents the number of migration loops in the graph, and $|V_r|+|E_r|$ is the complexity of finding migration loops of graph G_r.

5 Implementation

Hardware Switch. Our testbed implementation utilizes P4 hardware switches, specifically the Flnet S9180-32X with Intel Tofino ASIC [22], to model the hardware pipeline. The pipeline is composed of 12 stages, with each stage featuring 10Mbits of SRAM, 528Ktrits of TCAM, and 8 stateful ALUs. To account for the ASIC's dependencies, the pipeline defines the following dependencies: (1) match dependency, i.e., upstream table modifies downstream table; (2) action dependency, i.e., two tables mutually modify the same field; (3) successor dependency, i.e., the conditional operation.

Table control. Similar to work [26], *Monte* assigns a unique *SFC_id* and *NF_id* to the header of each packet, enabling the identification of packets from different SFCs and determining the specific logic that should process each packet. These two identities are added as keys to match in each match-action table, allowing table entries to match specific SFC packets. This enables dynamic manipulation of table entries during runtime to control the processing sequence of SFC packets and add or remove sharing nodes without incurring reconfiguration costs. Specifically, deleting a node logic of the CFG can be achieved by removing entries on runtime, without bringing reconfiguration cost.

Migration operations. In the reconfiguration operation, we utilized the fast reconfiguration technique (*fastreconfig*) provided by Intel Tofino hardware switches. This feature allows for rapid reconfiguration of different P4 programs within 50 milliseconds, including changes to the P4 logic and the insertion of new flow tables. In the state migration operation, we used the packet generator provided by the P4 switch, as implemented in P4Sync [14], to generate packets carrying state information. This generator produces a series of packets with incrementing hash values, allowing access to specific register locations in the switch. Similar to SwingState [13], we augmented each register in the P4 code with a simple logic to update and retrieve the register values. For interactions with the CPU, we specified a CPU port, which is accessed via a PCIe connection [22].

6 Evaluation

6.1 Experimental setup

Hardware and software environments. The hardware testbed consists of two P4 hardware switches (equipped with Intel Tofino ASIC) and two servers with 8 Intel Core i7-4771 CPUs running at 3.50GHz, connected through 10 Gbps fiber links. $P4_{16}$ programme language is used in the implementation.

In network simulations, we constructed environments using Mininet and P4 software switches with the Bmv2 target [27]. We tested two different network topologies: AboveNet [28], which has 22 nodes, and a FatTree-based [29] topology with a pod size of k=2 (i.e., each pod has four switches). A subset of switches are randomly selected as P4-programmable switches to implement SFCs, while the remaining switches acted as L3 routing switches. The latency of each link in the network ranged from 2ms to 20ms.

Migration rate. To evaluate the performance of *Monte* in handling SFC changes, we generate random SFC changes such as adding or deleting network functions, and modifying SFC attributes such as latency and throughput. Inspired by work [8], the migration rate Λ is proposed as a parameter to tune the degree of SFC changes. It is defined as the ratio of the number of affected network functions in the new SFC to the number of network functions in the original SFC.

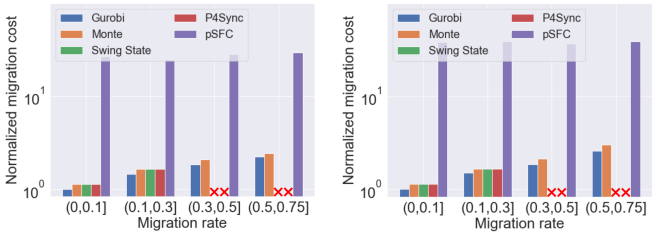
Migration downtime. Downtime is yielded by the two migration operations. In experiments, downtime of the reconfiguring operation is set to 50ms [22]. Downtime of the state migration operation approximately equals link latency because the amount of migrated states is under the capacity of the PCIe tunnel.

Construction of SFCs. We select network functions (i.e., P4 programs) from the P4 tutorial [30], example programs of the Intel Tofino ASIC [22], and stateful network function works [31]. We randomly generate five SFCs, each of which chains between 4 to 8 network functions, where the same network functions may appear in different SFCs.

Comparison methods. Two migration schemes are used for comparison, i.e., Swing State [13] and P4Sync [14]. However, they are designed for migration from a migrated target switch to a backup switch and lack strategies to arrange the migration within multiple switches.

A migration mathematical model is created as the baseline, whose optimal solution is obtained from Gurobi [32]. In the model, the migration process can be modeled from a given deployment to a migrated deployment, with binary variables to present node placement. Thus, the migration cost can be computed by changes in these binary variables after migration. However, achieving live migration in the migration model is challenging. We calculated the top-k optimal migration results from the solver as the optimal result for comparison with *Monte*.

We also compare *Monte* with pSFC [6], which is an SFCs deployment scheme in the programmable data plane that aims to minimize deployment cost while respecting constraints between hardware pipeline and network attributes. pSFC uses a heuristic algorithm and an ILP to make a trade-off between performance and execution time. Two heuristics are designed for deployment cost comparison, FFS (First Fit by Size) [23] and TS (Topological Sorting). Their main ideas are to generate the placing node sequence with different strategies: FFS priorities nodes with large resource usage and TS only generate a topological sequence of the CFG.



(a) FatTree (b) AboveNet

Fig. 4. Comparison of migration cost, where the normalized migration cost is calculated as the result of each bar divided by the result of Gurobi under migration rate ranging from 0 to 0.1.

6.2 Performance of migration optimization

Comparison of migration cost. We conducted an evaluation of *Monte*'s performance in migration. We started by generating five SFCs and deploying them using a deployment algorithm such as pSFC, which served as the initial state. We then randomly generated changes in the SFCs' network functions and attributes, with the degree of change varying based on the migration rate parameter Λ. As Λ increased, the possibility of changes in SFC attributes also increased. We ran more than 50 iterations for each range of Λ for the three methods and evaluated the migration cost. Two network topologies, FatTree and AboveNet, were built for migration evaluation, and a subset of nodes within these topologies (i.e., 7 nodes in FatTree and 11 nodes in AboveNet) was selected as available P4 programmable switches for SFC deployment and migration.

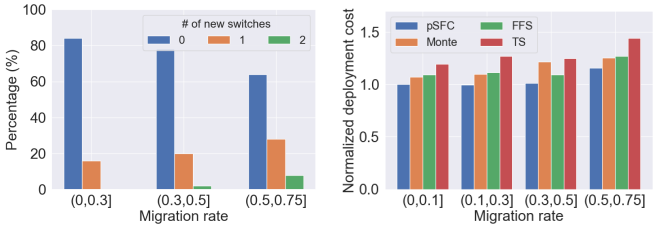
Results in Fig. 4 demonstrate that *Monte*'s migration cost is, on average, 94.03% lower than that of pSFC. As the value of Λ increases, both *Monte* and Gurobi incur higher migration costs, as they need to perform more reconfiguring operations and stateful migrations. However, pSFC's migration cost remains high due to its tendency to shuffle and rearrange nodes among network functions in the hardware pipeline even when there are only slight modifications to network functions. *Monte* achieves lower migration costs than pSFC because it adopts a different strategy to handle changes in SFCs. Specifically, pSFC frequently generates migration loops by rearranging nodes in the pipeline, whereas *Monte* focuses on maintaining the initial deployment state. As a result, *Monte* incurs fewer migration costs.

The Gurobi scheme, leveraging a larger search space, achieves superior optimization results compared to heuristic-based algorithms in *Monte*. In contrast, Swing State and P4Sync, designed primarily for single-switch scenarios, can only work in the migration of the single switch at lower migration rates, but become ineffective when the migration scheme involves multiple switches at higher migration rates. For comparison with *Monte* in Fig. 4, we removed the infrequent cases of failure from Swing State and P4Sync that required multiple switches for migration at lower migration rates (i.e., (0,0.1] and (0.1,0.3]).

Comparison of running time. The running time of the three methods is evaluated and presented in Table 2. Two

TABLE 2
Execution time (seconds).

Network setting	*Monte*	pSFC	Gurobi #1	Gurobi #2
FatTree (low λ)	0.187	0.131	26.40	∞
FatTree (high λ)	0.179	0.127	79.23	∞
AboveNet (low λ)	0.205	0.152	37.12	∞
AboveNet (high λ)	0.217	0.147	128.4	∞



(a) The number of new substrate switches when *Monte* outputs the minimal migration cost.

(b) Comparison of deployment cost.

Fig. 5. Balancing migration and deployment cost.

different objectives are set for Gurobi #1 and Gurobi #2 in the experiments. Gurobi #1 minimizes only the migration cost (e.g., $\omega = 1$ in Equation (15)), while Gurobi #2 balances both migration and deployment cost (e.g., $0 < \omega < 1$). Additionally, the *TimeLimit* parameter is set to 48 hours, and Gurobi will terminate the algorithm if it exceeds the time limit. However, the solutions may be sub-optimal.

The experimental results indicate that topology scale and migration rate can significantly affect the running time of Gurobi #1. As the size of topology expands, the search space also increases, resulting in a longer running time. Similarly, a higher migration rate leads to a larger number of nodes to be migrated, lengthening the search time.

In contrast to Gurobi, the model with consideration of the deployment objective is highly time-consuming as it increases the search area to balance migration cost and deployment performance simultaneously. Consequently, Gurobi #2 failed to find the optimal solution within the limited time.

In comparison, *Monte* is more practical as it can effectively respond to dynamic network requirements with an acceptable running time. In summary, *Monte*'s migration strategies are effective in reducing migration costs, allowing it to cope better with SFC dynamics, and it strikes a good balance between optimality and efficiency.

Comparison of migration and deployment cost. To investigate the trade-off between deployment and migration costs, we introduce a parameter h to control the number of new substrate switches that can be used for migrating SFCs and have not yet been used for placement. By allowing the algorithm to explore migration schemes that involve using these new switches, we can potentially achieve lower migration costs, as these switches can be pre-configured without disrupting the current SFCs deployment and then rerouted in live migration. However, a higher value of h also increases the chained length of deployed switches, which can potentially result in increased migration costs for subsequent migrations, as observed in our previous experiments. Furthermore, a longer chain of deployed switches also increases the risk of deployment failure if SFCs' latency requirements change, which could force the algorithm to find a new path to migrate, incurring large migration costs.

We also evaluate the number of new substrate switches required when *Monte* outputs the minimal migration cost result from all feasible schemes. In Fig. 5(a), the number of new substrate switches increases with the rise of the migration rate. The main reason is that deploying SFCs in new substrate switches is a good option to reduce migration cost when applying migration operations only in one switch cannot satisfy SFCs changes. Hence, we set a dynamic value to h which is always equal to 0 until there are no feasible solutions. Once all schemes with $n = 0$ fail, the algorithm starts to increase h until finding sufficient stage resources to place. In other words, the strategy of *Monte* tends to fill the deployed switches first for dealing with SFCs changes until all switches cannot satisfy the changes. Applying this strategy, we conducted the experiment in Fig. 5(b).

The deployment cost gap between *Monte* and pSFC is small. Benefits from fine-grained pipeline placement analysis, the deployment cost of *Monte* is superior to the other two general heuristic algorithms in most time. On the one hand, in pSFC, due to the existence of network function deletion in SFCs changes, sometimes migration cost reduces after migration. On the other hand, in *Monte*, since it should decrease the number of moving nodes, deployment cost always increases or remains unchanged. In short, *Monte* can use the parameter h to make the tradeoff between deployment and migration cost effectively.

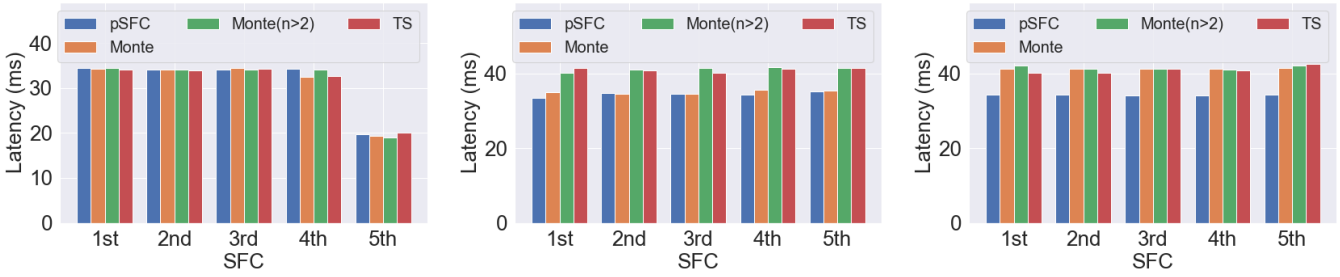
6.3 Performance of latency in Mininet

In this experiment, we measured end-to-end latency in a network simulator in different migration rates, e.g., (0, 0.3), (0.3, 0.5), and (0.5, 0.75). Given an initial deployment state in FatTree, we generated SFCs changes by the three migration rates and implemented the migration results derived from the three algorithms (Fig. 6). With a small Λ, the three methods have nearly the same latency because small changes in *Monte* can be directly tackled in idle stages. The fifth SFC has less latency since it is deployed on fewer switches. In a median Λ, average SFCs latency in *Monte* ($n > 2$) starts to increase. This is because it found and chose the migration scheme with less migration cost via deploying the new switch. Latency in TS starts to rise because of the increase in deployment cost, and it requires more switches to deploy. In the large Λ, pSFC can still compute the scheme with minimal deployment cost. However, *Monte* (dynamic h) cannot find feasible solutions to place new nodes of SFCs changes by its strategy of keeping minimal migration cost. Eventually, *Monte* with two different h settings requires the new switch for deployment.

In fact, the third condition rarely appears in our experiments. The worst case might exist because *Monte* brings extra deployment cost when it minimizes migration cost. Meanwhile, the deployment cost can affect the performance of SFC latency. Thus, the fine-grained pipeline placement strategy is essential in *Monte*.

6.4 Migration downtime in hardware switches

We built a topology using two P4 hardware switches as well as two servers to send end-to-end packets to measure



(a) Small λ (b) Median λ (c) Large λ

Fig. 6. Latency in Mininet.

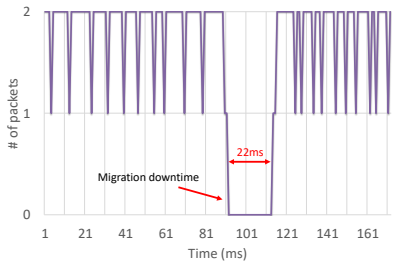


Fig. 7. Experiment on P4 hardware switches (Tofino ASIC).

downtime in migration. The migration request is generated at a random time point, and the controller produces the new P4 reconfiguration file via algorithms of *Monte*. Then, we cut down the process of the current switch and used the function *fastreconfig* that Tofino ASIC provides to replace the original file with the new reconfiguration file, where new entries are installed via the controller simultaneously.

Results in Fig. 7 show that the migration downtime time lasts for 22ms. Packet loss is inevitable during the downtime, and recovery of these packets can only rely on other methods (e.g., retransmission). Therefore, *Monte* is necessary because it can effectively mitigate the degradation of network performance resulting from migration in the programmable data plane.

7 Conclusions

This paper presents *Monte*, a scheme for migrating SFCs in the distributed programmable data plane. *Monte* formulates the migration process as an integer programming problem to optimize both migration and deployment cost. Moreover, *Monte* designs a live migration solution that minimizes migration cost in the dynamic migration process. Experimental results demonstrate that *Monte* significantly reduces migration cost while saving pipeline resources for SFCs deployment.

Acknowledgments

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant 62172189; the Innovate UK grants 10098627 and 10106629.

References

[1] Lin Cui, Fung Po Tso, Song Guo, Weijia Jia, Kaimin Wei, and Wei Zhao. Enabling heterogeneous network function chaining. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):842–854, 2018.

[2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[3] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881, 2021.

[4] Yu Zhou, Jun Bi, Cheng Zhang, Mingwei Xu, and Jinaping Wu. Flexmesh: Flexibly chaining network functions on programmable data planes at runtime. In *IFIP Networking Conference (Networking)*, pages 73–81. IEEE, 2020.

[5] Mu He, Arsany Basta, Andreas Blenk, Nemanja Deric, and Wolfgang Kellerer. P4NFV: An NFV architecture with flexible data plane reconfiguration. In *14th International Conference on Network and Service Management (CNSM)*, pages 90–98. IEEE, 2018.

[6] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. Compiling service function chains via fine-grained composition in the programmable data plane. *IEEE Transactions on Services Computing*, 2023.

[7] Tianzhang He and Rajkumar Buyya. A taxonomy of live migration management in cloud computing. *ACM Computing Surveys*, 56(3):1–33, 2023.

[8] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 651–665, 2022.

[9] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 635–649, 2022.

[10] Yiming Qiu, Ryan Beckett, and Ang Chen. Synthesizing runtime programmable switch updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 613–628, 2023.

[11] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1289–1305, 2022.

[12] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, TS Eugene Ng, et al. A vision for runtime programmable networks. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, pages 91–98, 2021.

[13] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing state: Consistent updates for stateful and programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 115–121, 2017.

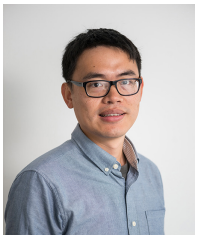
[14] Jiarong Xing, Ang Chen, and TS Eugene Ng. Secure state migration in the data plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 28–34, 2020.

[15] Hongyi Huang, Wenfei Wu, Yongchao He, and Zehua Guo. SFP: Service function chain provision on programmable switches for

cloud tenants. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1239–1249. IEEE, 2022.
[16] Tomasz Osiński, Halina Tarasiuk, Lukasz Rajewski, and Emil Kowalczyk. DPPx: A P4-based data plane programmability and exposure framework to enhance NFV services. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 296–300. IEEE, 2019.
[17] Junte Ma, Sihao Xie, and Jin Zhao. P4SFC: Service function chain offloading with programmable switches. In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6. IEEE, 2020.
[18] Xiang Chen, Qun Huang, Peiqiao Wang, Zili Meng, Hongyan Liu, Yuxin Chen, Dong Zhang, Haifeng Zhou, Boyang Zhou, and Chunming Wu. LightNF: Simplifying network function offloading in programmable networks. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.
[19] Jaewook Lee, Haneul Ko, Hochan Lee, and Sangheon Pack. Flow-aware service function embedding algorithm in programmable data plane. *IEEE Access*, 9:6113–6121, 2020.
[20] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated service chaining on a single switch asic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 141–149, 2019.
[21] Xiang Chen, Dong Zhang, Xiaojun Wang, Kai Zhu, and Haifeng Zhou. P4sc: Towards high-performance service function chain implementation on the p4-capable device. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1–9. IEEE, 2019.
[22] Intel Tofino switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html. Accessed on: Sep. 8, 2024.
[23] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, 2015.
[24] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-efficient and high-performance deployment for data plane programs. In *IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2020.
[25] Gang Sun, Zhu Xu, Hongfang Yu, Xi Chen, Victor Chang, and Athanasios V Vasilakos. Low-latency and resource-efficient service function chaining orchestration in network function virtualization. *IEEE Internet of Things Journal*, 7(7):5760–5772, 2019.
[26] Junte Ma, Sihao Xie, and Jin Zhao. Flexible offloading of service function chains to programmable switches. *IEEE Transactions on Services Computing*, 16(2):1198–1211, 2022.
[27] p4lang/behavioral-model. https://github.com/p4lang/behavioral-model. Accessed on: Sep. 8, 2024.
[28] Abovenet. http://www.topology-zoo.org/maps/Abvt.jpg. Accessed on: Sep. 8, 2024.
[29] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.
[30] p4lang/tutorials. https://github.com/p4lang/tutorials. Accessed on: Sep. 8, 2024.
[31] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM Conference*, pages 29–43, 2016.
[32] Gurobi - The Fastest Solver. https://www.gurobi.com/. Accessed on: Sep. 8, 2024.

Xiaoquan Zhang is currently a postdoctoral researcher in the College of Cyber Security at Jinan University, Guangzhou, China. He received the Ph.D. degree from Jinan University, China, in 2024. His current research interests include programmable data planes, software-defined networking, and machine learning in computer networks.

Lin Cui is currently a professor in the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on software defined networking (SDN), programmable networking, NFV, cloud networking, distributed systems and so on.



Fung Po Tso received his B.Eng., M.Phil. and Ph.D. degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently a senior lecturer in the Department of Computer Science at the Loughborough University. His research interests include: network policy management, network measurement and optimisation, data centre networking, software defined networking (SDN), and edge computing.

Yuhui Deng received the Ph.D. degree from the Huazhong University of Science and Technology in 2004. He is currently a Professor with the Computer Science Department, Jinan University. He worked with the EMC as a Senior Research Scientist from 2008 to 2009 and Cranfield University from 2005 to 2008. His research interests cover green computing, cloud computing, information storage and computer architecture.

Zhetao Li is a professor and dean of the College of Information Science and Technology, Jinan University. He received the B.Eng. degree from Xiangtan University in 2002, M.Eng. degree from Beihang University in 2005, the Ph.D. degree from Hunan University in 2010. His research interests include cloud computing, artificial intelligence, and multimedia signal processing.

Weijia Jia is currently a Chair Professor and Director of BNU-UIC Institute of Artificial Intelligence and Future Networks. His research interests include smart city, IoT, knowledge graph constructions, multicast and anycast QoS routing protocols. He has over 500 publications in prestigious international journals/conferences and research books.