DINC: Toward Distributed In-Network Computing

CHANGGANG ZHENG, University of Oxford, UK
HAOYUE TANG, Yale University, USA
MINGYUAN ZANG, Technical University of Denmark, Denmark
XINPENG HONG, University of Oxford, UK
AOSONG FENG, Yale University, USA
LEANDROS TASSIULAS, Yale University, USA
NOA ZILBERMAN, University of Oxford, UK

In-network computing provides significant performance benefits, load reduction, and power savings. Still, an in-network service's functionality is strictly limited to a single hardware device. Research has focused on enabling on-device functionality, with limited consideration to distributed in-network computing. This paper explores the applicability of distributed computing to in-network computing. We present DINC, a framework enabling distributed in-network computing, generating deployment strategies, overcoming resource constraints and providing functionality guarantees across a network. It uses multi-objective optimization to provide a deployment strategy, slicing P4 programs accordingly. DINC was evaluated using seven different workloads on both data center and wide-area network topologies, demonstrating feasibility and scalability, providing efficient distribution plans within seconds.

1 INTRODUCTION

In-network computing services, offloading applications to the programmable data plane, have been explored for purposes such as caching [25], inference [7, 29], and compression [45]. Its benefits include line-rate throughput, lower latency, power efficiency and data reduction.

However, scaling in-network services is hard, as programmable network devices are intended for high-efficiency packet processing, with limited resources (e.g., memory, operations, and stages) compared with CPUs. One approach is to optimize algorithms' design for a single-device deployment, yet resource constraints remain a limitation. An alternative solution is moving to *distributed* in-network computing, jointly utilizing resources of programmable network devices.

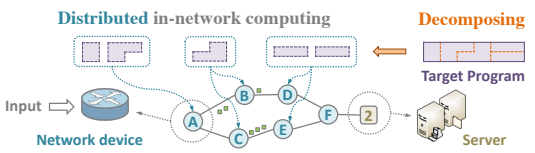


Fig. 1. Distributed in-network computing paradigm.

Distributed in-network computing raises multiple implementation challenges, especially where resource-heavy applications are considered, such as large machine learning (ML) models. Figure 1 illustrates the challenges: (1) **Decomposing** the single program into multiple *segments*. (2) **Distributing** the program's segments across multiple devices without affecting the correctness of its functionality. (3) Satisfying the program's and network's set of **constraints**, such as latency and resource constraints. (4) Providing the program's functionality for any set of paths within the network without routing rules changes. This last challenge is possibly the hardest, as in a network packets may travel from any node to any node, and operators may use different routing optimization methods.

In a software-defined network, a controller has a centralized view of the network, including device information, and potential routes through the network. Building upon this information, a controller can be designed to provide a joint resource provisioning plan for distributed in-network

C. Zheng, X. Hong, and N. Zilberman, e-mail: {name.surname}@eng.ox.ac.uk, are with the University of Oxford, UK; H. Tang, A. Feng, and L. Tassiulas, e-mail: {name.surname}@yale.edu, are with the Yale University, New Haven, USA; M. Zang, e-mail: minza@dtu.dk, is with the Technical University of Denmark, Denmark.

computing. It can utilize unused network resources, splitting a single in-network computing service across several devices.

Network service chains provide heuristic solutions for segment placement and distributed planning [47]. However, there are intrinsic and significant differences between it and distributed in-network computing, both in terms of the type of devices used and their location. Unlike CPU- and GPU-based service chains, deployed at the server level, in-network computing service segments are deployed physically within the network, on switch-ASIC [5], FPGA [20], or network interface cards (NICs) [44]. The architecture, resource constraints, communication models and performance requirements are inherently different to traditional service chains (see §2).

Preliminary efforts toward distributed in-network computing [8, 9, 32, 43] focused on the distribution problem. Some of these works [8, 9] distributed multiple programs, rather than slicing a single program, or partitioned Match-Action tables using manual directives [32]. Flightplan [43] was the only one to disaggregate and place a distributed program. Yet little effort has attended to the challenge of routing through the network, from any node to any node, without routing rules modifications or at the scale of a wide area network (WAN).

In this paper, we present DINC, Distributed In-Network Computing. DINC efficiently plans and implements P4-based service partitions on multiple network devices. DINC's planner supports any-to-any routing and is able to distribute and deploy program segments across network devices while providing full, correct functionality. To ensure co-existence with normal network functionality, DINC co-designs a code slicer and generator, extracting and generating P4 program slices in accordance with the planner's strategy. DINC is a scalable, flexible, and easy-to-deploy modular framework. In summary, our key contributions are:

- Introducing a distributed in-network computing solution for in-network services.
- Developing a mathematical model for distributed in-network computing programs deployment within many-paths networks. The model is simple, efficient, and extendable.
- Presenting a mechanism ensuring the full execution of each program.
- Implementing DINC, a modular framework for slicing data plane programs and deploying them on a distributed set of programmable network devices. DINC is easily extensible with new deployment strategies, devices, and architectures, and co-exists with normal network functionality without changes to routing rules.
- Demonstrating the applicability for three in-network computing services (load-balancing, telemetry and ML), showing scalability to large networks. These are prototyped and evaluated both on a switch-ASIC and in an emulation environment, using a large ISP (Internet service provider) wide area network and Folded-Clos data center topologies.

2 MOTIVATION

Programmable network devices are resource-constrained due to their performance-driven design logic [17]. For example, Intel Tofino switch [16] can guarantee Tbps-scale throughput, but has only 12 processing stages and Mb-scale memory. These resource constraints are a main challenge for offloading applications to the network. Still, in-network computing prototypes (e.g., caching [25], aggregation [28, 40], ML inference [48, 57], resource scheduling [4], and consensus [13]) demonstrated real-time processing, offloading computing tasks from servers and achieving high performance.

While the performance on a single device is high, resource contention between in-network computing applications and network functionality is a barrier for adoption. To illustrate the challenge, Figure 2 shows the resource consumption of the RARE open source router [33] when deployed on Tofino. As shown in Figure 2, the router consumes more than half of the pipeline stages with only basic functionality. Using more features of the RARE router exhausts and even exceeds Tofino's resources.

Deploying an in-network computing program on the same device as RARE, is hard or impossible without stage sharing. Figure 3(a) shows the feasibility of deploying example ML models (using Planter [57]) on Tofino, standalone and coexisting with RARE. As a model's size increases to improve its performance, resource consumption increases too, as shown in Figure 3(b).

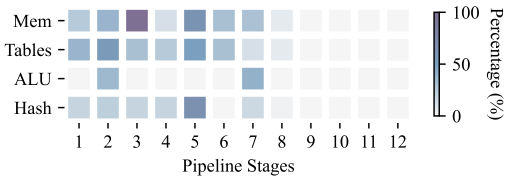
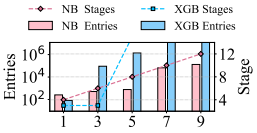


Fig. 2. Resource consumption of a RARE Router with basic functionality on Intel Tofino. The color bar indicates the percentage of utilized resources.

Types	SVM	NB	RF	XGB
Stages	9	8	6	15
Entries	0.1M	0.1M	0.3M	1.4M
Alone	✓	✓	✓	✗
Coexist	✗	✗	✗	✗

(a) Alone vs Coexist



(b) Parameter vs Resources

Fig. 3. ML models (a) resource consumption and (b) scaling with model's hyper-parameters. *NB* - Naïve Bayes, SVM - Support Vector Machine, RF - Random Forest, XBG - XGBoost. The hyper-parameter used for XGB is depth and for NB is number of features.

Consider the case where the ML models shown in Figure 3 are used to implement a cyber-security service, detecting and dropping malicious traffic. In a wide area network (WAN), such malicious traffic may come from any user, pass through any switch, and go to any destination. The cyber-security service will need to be deployed in a manner that guarantees that no matter the path taken through the network, malicious packets will be detected and dropped.

To support *both* normal network functionality, and the ML-based service, distributing programs across several devices is needed. However, it is not easy to distribute an in-network application across multiple devices, for the following reasons: 1. There is no agreed model of the network used for distributed in-network computing. Intuitively, in-network computing should not affect existing network functions, nor the routing rules used to forward packets. 2. After program segmentation, there are parameters shared across segments of the application, and the data passing model is undefined. 3. Application splitting and coexistence with network functions is error-prone, as well as guaranteeing segmentation execution order. To address these challenges, this work explores efficient methods for deploying in-network computing services in a distributed manner.

3 THE CONCEPT OF DINC

To explain how in-network computing can be distributed and deployed, we first discuss common properties of algorithms using an example in-network computing application, and then demonstrate the many-paths nature of the network using a sample network topology.

3.1 In-network Computing Example

In-network computing provides application functionality by mapping computation tasks to programmable data planes on network devices. These data planes use match-action pipelines, where values are looked-up in a table, and the result of the lookup is an action. Typically, every match-action pair consume a processing state within the pipeline. Sequential dependencies between operations lead to a series of stages used on the device, with metadata used to pass shared information between stages (metadata is stored in a packet header vector (PHV) [5], which is initialized per packet). Despite the high performance of existing work [2, 13, 18, 25, 28, 51–53, 57], resource limitations remain a constraint. As demonstrated in Figure 3, scaling up computing complexity leads to exhaustion of resources. We use Naive Bayes (NB) as an example of a classical ML classification algorithm that uses typical in-network ML mapping and faces resource constraints.

Equation 1 shows the in-network Bayes mapping used by [57]. Different from traditional Naive Bayes, the log(#) operation converts multiplication into addition (as multiplication is not supported

on a switch). The map(#) operation ensures that used intermediate values are covered with minimal accuracy loss.

$$\hat{y} = \arg\max_{y}[\text{map}(\log_2 P(y)) + \sum_{i=1}^{n} \text{map}(\log_2 P(x_i \mid y))] \quad (1)$$

Figure 4 shows the data plane realization of the above equation. Step ❶ shows the extraction of n features (fields) from the packet header, and storing them in metadata fields as f_{x1} to f_{xn}. Next, the probability of each class $\text{map}(\log_2 P(y))$ is read from a table called *read probability (RD Prob)* into metadata fields as m_{c1} to m_{cm} (given a classification problem with m classes) in Step ❷. For every input feature i, Steps ❸ and ❹, look up an intermediate value $\text{map}(\log_2 P(f_{xi} \mid cj))$ and add it to its respective class j. The prediction probability of all m classes will be m_{c1} to m_{cm}, and the final pipeline stage (marked *compare*) finds the class with the maximum probability through comparison, and sets it as the output label (Step ❺).

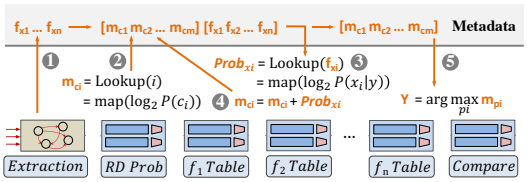


Fig. 4. Data plane implementation of Bayes based on [57] and Equation 1.

While mapping details vary between applications, this example is representative in terms of its common stage-based structure and metadata passing. The distributed deployment strategy of this example on a sample network topology, and the challenges it faces, will be explained in Section 3.3.

3.2 Network Scenario

In this work, we consider a distributed in-network computing deployment scenario in a network with many-paths, using multiple ingress and egress nodes. Data can originate from any ingress node and terminate at one or more egress destinations. Beyond this any-to-any or many-to-many connectivity, techniques such as load balancing or routing redundancy mean packets from the same service may be routed through multiple paths, for a given source-destination pair.

Distributed in-network computing should not affect existing network protocols (e.g., IS-IS) nor services (e.g., load balancing). Given a deployment scenario and underlying routing rules, distributed in-network computing should find best-effort service deployment within the given network constraints, without changes to routing rules. In this manner, packets from any source node should be fully processed before reaching their destination along any possible path.

3.3 Distributed Deployment Example

DINC enables the deployment of large in-network computing programs, as its framework is able to slice a program into segments and deploy these segments within network devices given resource constraints (e.g. memory, operations, stages).

Figure 5 shows a sample deployment of a Bayes classification algorithm (explained in Figure 4) using three-features on a Folded-Clos topology (assuming all network devices are programmable). Two inputs (core switches) and four outputs (edge switches) are assumed[1]. Packets can through any downstream path with minimal changes. Segments on each device are shared among multiple paths flowing through the node.

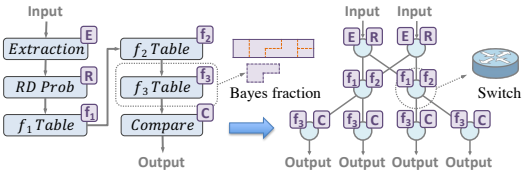


Fig. 5. An example deployment of an in-network algorithm (Bayes) on network topology (Folded-Clos).

[1]This is a simplified scenario, traffic is presumably generated outside this network.

This figure shows an ideal example of a distributed deployment, where *Extraction (E)* and *read probability (RD Prob)* are deployed in the first hop (core switches), *feature tables 1 & 2 (f_1, f_2)* in the second hop, and *feature tables 3 (f_3)* and *Compare (C)* in the last hop (edge switches). The deployment may be obvious, as the network is symmetric and well structured. For larger or more complex networks and programs, the deployment is complicated, as demonstrated in §9.

4 DINC OVERVIEW

To provide an ideal deployment of in-network computing program segments on complex network topologies, DINC answers three key questions:

- *How to plan and distribute segments across multiple paths in a given network?* (§5)
- *How to ensure functionality when programs are distributed inside a network?* (§6)
- *How to make distributed in-network computing easy to deploy?* (§7)

Before answering these questions, we first provide an overview of the DINC framework's operation, shown in Figure 6. DINC is given an in-network computing program with both data plane and control plane code components (shown in ❶), and a network topology (shown in ❷). DINC's P4 slicer (§7.1) extracts the program resource requirements (shown in ❻), dependencies (shown in ❸), and metadata information. The network controller provides the routing table, with all paths identified either by the controller or DINC (shown in ❹) and the resources available on each network device (shown in ❺). DINC's planner (§5) uses the outputs from steps ❸ to ❻ to craft an integer linear programming (ILP) problem and outputs a deployment strategy in step ❼. This guides the P4 generator (§7.2) for data plane and control plane codes generation in step ❽.

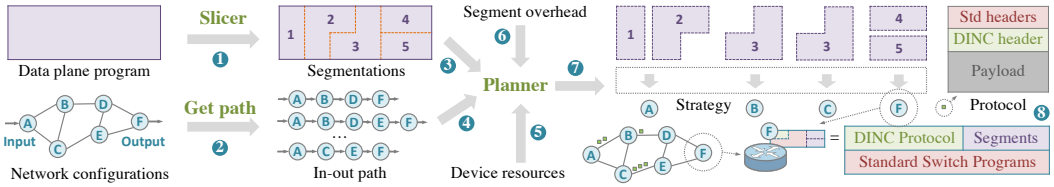


Fig. 6. DINC workflow overview (steps ❶ to ❽).

5 PLANNING

Most previous distributed in-network computing works focused on single-planned paths [12, 26, 46]. In this section, we address the absence of distributed planning for in-network computing across *multiple paths*, without influencing the routing rules of the original network. To tackle this, the problem is formulated within the DINC planner as an ILP problem and we introduce a solution with reduced complexity. In subsection 5.1, we introduce the network model and algorithm partition and formally define a deployment strategy. Subsections 5.2 and 5.3 model the deployment optimization mathematically into an integer linear programming problem. An ILP solver is given in the subsection 5.4. Terms used in this section are summarized in Appendix B.

5.1 Network Model

We focus on the in-network computing tasks planning problem among the programmable devices and hence ignore undeployable nodes after edge contraction. A set of network devices with order and without duplication along a data trace connecting the input and output nodes is referred to as an *In-Out Path*. A network with N_d deployable devices can be represented by tuple $(\mathcal{D}, \mathcal{P})$, where $\mathcal{D} := \{1, \cdots, N_d\}$ is the set of devices and $\mathcal{P}$ contains $N_p := |\mathcal{P}|$ paths. Each element $P_i \in \mathcal{P}, \forall i = 1, \cdots, N_p$ is an ordered set of size l_i, i.e., $P_i = \{p_i^1, \cdots, p_i^{l_i}\}$. The chain $p_i^1 \rightarrow \cdots \rightarrow p_i^{l_i}$

represents a path from an input device p_i^1 to an output device $p_i^{l_i}$, where l_i is the total number of devices in path i. There are N_r types of resources in the programmable device networks (e.g., storage), and we use R_d^r to denote the available resource type $r \in [N_r]$ on device $d \in \mathcal{D}$.

Assume that the target in-network computing algorithm can be decomposed into N_e elements. Let $\mathcal{E} := \{1, 2, \cdots, N_e\}$ be the set of all the algorithm's element indices. For any DINC planner, let $X_{e\to d} \in \{0, 1\}, \forall e, d$ be the deployment decision (strategy), where $X_{e\to d} = 1$ indicates algorithm element e is deployed on device d. If $X_{e\to d} = 1$, by running element e, device d will cost O_e^r units of resource r (e.g., CPU running time or storage).

The goal is to design a deployment strategy, i.e., $\{X_{e\to d}\}$, that can achieve application objectives (e.g., low latency) and a small number of duplicated deployed segments while consuming little resources on the programmable devices.

5.2 Constraints

A successful deployment strategy should typically satisfy dependency, integrity, and resource constraints, which are explained as follows:

5.2.1 Dependency: Assume that the N_e elements have to be completed in order among each *in-out path*. For every path, any elements e in the in-network computing algorithm should appear at least once before its successor e'. Mathematically, if element e' is deployed on device p_i^j, i.e., the j-th device of the i-th path, the deployment decision variable $X_{e\to p_i^j} = 1$ and element e has to be deployed on at least one node in set $\{p_i^1, \cdots, p_i^{j-1}\}$, i.e.,

$$\sum_{k=1}^{j-1} X_{e\to p_i^k} \geq X_{e'\to p_i^j}, \forall e < e', i \in [N_p], j \in [l_i], \tag{2}$$

5.2.2 Integrity: All the segments should be executed on each *in-out path* to satisfy the integrity of the algorithm. Therefore, on every path i, every element $e \in \mathcal{E}$ should appear at least once, i.e.,

$$\sum_{k=1}^{l_i} X_{e\to p_i^k} \geq 1, \forall i \in [N_p]. \tag{3}$$

5.2.3 Resource Constraints: The type r resource on each device d must be sufficient for all deployed elements. Therefore,

$$\sum_{e\in E} O_e^r X_{e\to d} \leq R_d^r . \forall d \in [N_d], r \in [N_r]. \tag{4}$$

5.2.4 Other Potential Constraints: The above constraints can be extended, encompassing additional and customized requirements, such as limitations on hops and latency for specific services, constraints on throughput, and variations in resource constraints using heterogeneous devices. The specific constraint conditions applied can be adjusted based on the specific deployment scenario.

5.3 Objective

We expect our deployment strategy to optimize the following three major objectives in distributed in-network computing: minimize the resource consumption, minimize the computation delay and minimize duplicate deployed segments. Quantifying the aforementioned objectives is as follows:

5.3.1 Resource Consumption. A good *deployment strategy* should minimize the total resource consumption of all types of resource r. Let $\mathbb{O}_{\mathrm{R},r}$ be the total type r resource cost in the network. Recall that O_e^r is the type r resource overhead if element e is deployed, for each deployment strategy

$\{X_{e\to d}\}$, then $\mathbb{O}_{R,r}$ can be computed as follows:

$$\mathbb{O}_{R,r} := \sum_{d=1}^{N_d}\sum_{e=1}^{N_e} O_e^r X_{e\to d}. \tag{5}$$

5.3.2 Latency. Assume that the transmission delay on each path is fixed regardless of the execution task. We then focus (as an example objective) on minimizing the execution latency. The execution latency on each in-out path $j \in [N_p]$ can be computed by accumulating the execution time of each element e. Suppose L_d^e is the execution time of element e deployed on device d. The execution delay $\mathbb{O}_{L,i}$ on path i can be computed by:

$$\mathbb{O}_{L,i} = \sum_{j=1}^{l_i}\sum_{e=1}^{N_e} L_e^{p_i^j} X_{e\to p_i^j} \tag{6}$$

5.3.3 Multi-objective Optimization. We aim at designing a deployment strategy that can optimize resource consumption and performance objectives (e.g., latency) at the same time. Such a problem can be formulated as a multi-objective optimization problem, where there may exist multiple Pareto optimal points (i.e., the execution latency cannot be minimized without consuming fewer resources). To find such Pareto optimal points, we took a linear scalarization approach [19]. The ultimate objective function is a weighted linear combination of the execution latency on all in-out paths and all types of resource consumption. i.e.,

$$\mathbb{O} := w_R \sum_{r=1}^{N_r} \mathbb{O}_{R,r} + w_L \sum_{i=1}^{N_p} \mathbb{O}_{L,i}, \tag{7}$$

where $w_R, w_L \in \mathbb{R}^+$ are the weights of resource consumption and execution latency, respectively. The above optimization objectives can be flexibly adjusted based on varying service requirements and deployment environments.

5.4 ILP Solver

Recall that deployment decision $X_{e\to d}$ are binary decision variables. Minimize the scalarized objective function (7) under constraints (2), (3) and (4) can be rewritten as an integer linear programming problem (ILP). The constraints (§ 5.2) and objectives (§ 5.3) in this section are the most common ones and can be amended or replaced (§ 7) for different applications.

Searching for the optimum solution of an ILP is NP-hard. We tackle this problem using the branch-and-cut method [36] that combines the cutting plane method and brand-and-bound method. The solving details can be found in Appendix C.

6 DISTRIBUTING SEGMENTS TO NODES

The previous section presented a theoretical solution in DINC for planning the program distribution strategy, while questions still remain on how to effectively deploy the program element into a programmable network topology based on the *deployment strategy*. In order to clearly address these questions, we provide a brief introduction to how typical in-network computing programs are executed. Figure 7 shows a sample in-network computing program. When packets come in, features f_1 and f_2 are extracted from the packet header to packet header vector (PHV) as metadata. Together with m_1, m_2, and f_3, all metadata conveys intermediate results between segments e_1, e_2, and e_3, where the output is feature f_3. The output $f3$ from the last segment e_3 is put back into the packet header as a result. Through this process, PHV will be cleared between packets and metadata will reset. For the distributed deployment of such a sample program, each segmentation should be

sequentially executed and each segment requires metadata from dependent segments in order to function correctly.

To realize this, two key questions should be addressed. 1. *How to encode and pass metadata among segments?* 2. *How to ensure all program elements are being executed in order and without duplication?* These two questions are solved by using the DINC header (§ 6.3), which is a dedicated header for distributed in-network computing algorithms.

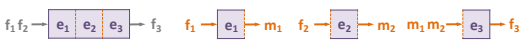


Fig. 7. In-network computing program, encode-based ML model as an example [57], and the sliced segments of the program.

A simple toy example shown in Figure 7 and Figure 8 is used to better demonstrate these two problems. Considering given constraints when deploying this algorithm in programmable network devices, one possible *deployment strategy* is plotted in Figure 8. In this case, there are three *in-out paths* within the topology $\{\{B\}, \{B \rightarrow C\}, \{A \rightarrow C\}\}$. The two problems addressed in this case are:

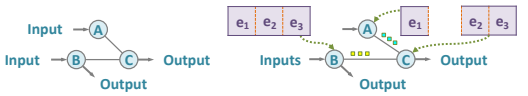


Fig. 8. The sample in-network computing programs deployment on the programmable network with two *in-out path*.

1. Along path $\{A \rightarrow C\}$, metadata from segment e_1 on device A should be able to be well encoded and passed to device C for element e_2 and e_3. At the same time, besides the input and output of element e_1, the system should recognize f_2 (input of e_2 on device C) as part of the input and output on device A. (§ 6.1)
2. Along path $\{B \rightarrow C\}$, after packets execute e_1, e_2, and e_3 on device B, the element e_2 and e_3 on device C should not be activated. While these two elements should be activated when it receives the packet along the path $\{A \rightarrow C\}$ from the device A. (§ 6.2)

6.1 Metadata Passing

Any used metadata should be well encoded and passed to the target segmentation from its predecessor segmentation. There are several options to convey information from one node to the other. The first option is to mirror and send the packet independently [1]. However, this option is not realistic without synchronizing arrival time and will not give a guaranteed metadata delivery. Another option is to pass through the control plane, the control plane cannot guarantee conveying of metadata and has a limited bandwidth of data transfer. In DINC, we choose to store metadata in a predefined header. Although a larger header will increase the average packet size and influence the packet rate, the evaluation shows the influence is minor and the program generated by DINC is able to reach the full line rate. Most importantly, it can guarantee the runtime metadata passing. Once the metadata are marked as the output of the segment, DINC will break and store it in several 32 bits chunks. In the parser of the following dependent segment, the auto-generated encapsulate logic will help to restore the metadata. To guarantee the consistency of metadata passing over segments, and ensure transmission reliability when it is not used in some of the segments, DINC designed a checking process to auto-complement missing metadata on markers.

6.2 Segment Traversing

In the distributed scenario, it is possible that segments are being duplicated especially when multiple paths join together as shown in Figure 8. To ensure every segment is executed once and only once, a bit map is held by each packet. Before the execution of any segmentation, it will check if the packet is fully processed by the predecessor-dependent segments. Once passing this check, along with the execution of the segment, the bitmap will be updated with the current segment. With the help of this per-segment bitmap checking process, no matter which path the packet comes from

or which set of segments the packet has been passed, they can share the same set of segments without being duplicated and executed. All the checking processes and segment bitmap IDs are auto-generated by the DINC framework.

6.3 DINC Header Design

No matter what the type of *deployment strategy* is, if there are intermediate metadata traversing between segments, data and segment information should be able to traverse among devices. Thus, a standard DINC protocol header is designed to meet requirements and allow services like intra-metadata encoding and passing (§ 6.1) and segment traversing (§ 6.2).

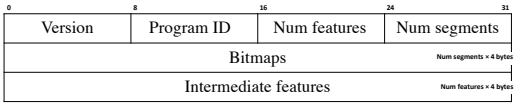


Fig. 9. The DINC header design.

As shown in Figure 9, beyond the DINC version, the protocol contains three types of information: 1. program related, 2. metadata related, and 3. segment related. The program-related information shows which one or several in-network computing programs are currently executed and included in the packet, mainly based on the Program ID field. The metadata-related contains information on the number of intermediate features included and their values. The bitmap information indicates the set of segments already executed. The overhead of DINC is limited to 16B if the program is sliced into 32 segments or less, and the metadata traversing between two segments is within 8B. Given the header in Figure 9 was designed with redundancy for future extensions (e.g., version, number of features, number of segments, 32 bits bitmap), the used DINC header can be smaller. The header size can be further minimized by adding constraints and objectives in the planner [9].

7 DINC FRAMEWORK DESIGN

The deployment of distributed in-network computing is challenging for two reasons. The first is the complexity of the data plane program. It is hard to correctly extract information from a given data plane program according to user expectations. The second is the complexity of data plane program generation. Reconstruction of a valid data plane program under a certain architecture, coexisting with its use case, and with an embedded DINC header is difficult, especially when the slice is complex.

The DINC framework is designed to tackle these two challenges. DINC contains a data plane program Slicer, a strategy Planner, a code Generator, and a Tester, allowing the generation of sliced data plane programs and their deployment on target hardware nodes. The detailed workflow of the framework shown in Figure 10 is as follows: ❶ *User input:* The target data plane program and slicer markers are required as user inputs for DINC configurations. ❷ *Network configuration:* The network topology and its related resources are required and stored in DINC configurations, which can generate directly from the network controller or emulate by using the DINC framework. ❸ *DINC slicer:* The network slicer slices the target program based on the markers, builds the dependency between each segment, and digs the resource information of each segment. ❹ *DINC planner:* The planner is used to generate deployment strategies for the target program. Based on the input network topology, segment dependency, remaining resources for each network device, and the resource overhead of each segment from previous steps, the planner applies the ILP (§ 5) to formulate the plan. ❺ *P4 generator:* The code generator generates data plane codes for all programmable network devices. To generate the codes, the generator jointly combines the selected architecture and use case based on the DINC configurations, and combines the program segments of the input program from the deployment strategy. ❻ *Synthetic test:* Before loading the generated code to the real hardware, a synthetic test is implemented by the framework. The framework will activate a test environment with the same topology as the input network, and load the generated

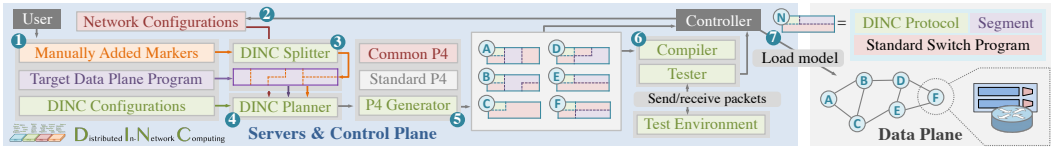


Fig. 10. The DINC framework components and workflow steps (❶ to ❼).

model for each node. The test can then be easily done within the environment while the content varies by use cases. P4 debugging tools can be used in the field to trace and correct bugs in each generated segment [42, 59]. ❼ *Segment loader:* When the test is finished, the generated data plane programs with sliced segments are to be sent to the controller for deployment.

Among the seven key components mentioned above, the DINC planner has been introduced in Section 5. The other two key components that simplify the process of distributing segments to nodes are introduced: the DINC specialized P4 code slicer (§ 7.1) and P4 code generator (§ 7.2). The rest four components are used as auxiliaries for the evaluation and functionality tests where the results will be presented in Section 9.

7.1 Code Slicer

The DINC slicer is used to segment the input data plane program and extract information from it. There are two design options, the first is a compiler-like module that is able to auto-detect and extract information. Despite its good intentions, this design needs extra configuration efforts, limits the scope of input programs and information types and can lead to errors. The second, used by DINC's slicer, is to use manually added markers to slice the program and extract the information. This solution is lightweight and flexible.

A functional slicer needs to support several properties of the program: 1) The code is between different parts of the pipeline (e.g., parser, ingress, egress). 2) The dependency of each segment is complex. 3) The required resources may change between cases.

Manually added markers enable solving these challenges. We use a fragment of a sample-sliced program in Table 1 to show how it works. In the program, markers @! and !@ are used to notify the slicer where the information should be extracted. The marker design is flexible and can be customized in the framework. The markers are written in comments and do not affect the execution of the original program. Further information related to slicer design is provided in Appendix D.

```
1  control Ingress(...){
2      // @!S,0!@ @!P,control!@
3      table e_1{...}
4      action e_1(...){...}
5      // @!E,0!@
6      // @!S,1!@ @!P,control!@
7      table/action...
8      // @!E,1!@
9      apply{
10     // @!S,0!@ @!Pre,none!@ \
11        @!P,control-apply!@
12         e_1.apply();
13         // @!E,0!@
14         // @!S,1!@ @!Pre,none!@
           ↪ @!P,control-apply!@
15         e_2.apply();
16         // @!E,1!@
17 }   }
```

Table 1. Sample in-network ML P4 code (DT) with markers to show a sliced pattern as in Figure 7. Segment information like End, Slice, Previous, and Position are included between markers @! and !@.

7.2 Code Generator

The design objective for the code generator is to generate the data plane program that can be directly applied to each node. The key problem is *How can we most reuse the resources during the generation process?*

To solve this, the DINC generator is designed based on a controller plus multiple architectures and uses case building blocks. Before generating any node, the controller will invoke the selected

architecture and use case block based on the network controller or the DINC configurations. Besides writing the skeleton of the data plane program, the selected architecture folder will call the respective use case function for writing use case codes at each position of the program. The deployment strategy will also be applied in the architecture block to drive the regeneration of the sliced segment at the right position. Functions that can auto-generate P4 codes dealing with metadata embedding and extraction as well as bitmap checks are applied and can be called by any nodes or segments during the generation process.

7.3 Modular Framework Design

The new architecture and target design appear frequently, and a design that allows iteration is required. The DINC framework applies a modular design, where a centralized controller is used to provoke corresponding modules according to the input DINC configurations. When a module is selected, all the functions under this module will be loaded to the controller. DINC currently are designed to support modular topology generators, solvers, architectures, targets, slicers, use case generators, and testers. With this modular framework design, DINC can be adapted to the required case easier and with strong customized capacity.

7.4 Multi-Program Deployment and Resource Fairness

DINC is primarily designed to distribute a single large program across multiple devices when it cannot fit on a single device. However, it does not currently support the automatic distribution of multiple programs provided incrementally; it only supports iterative deployment. To ensure fairness, limitations should be imposed within the planner, such as allocating available resources evenly or adding upper constraints on the total resources for each resource type. Further elaboration related to fairness can be found in Appendix E.

8 IMPLEMENTATION

The DINC framework is implemented in Python 3.10. The ILP solver is based on *milp*, a mixed-integer linear programming solver in *SciPy v1.10.0*. The topology is stored using *NetworkX 3.0*. The framework is open and available on the DINC's GitHub repository [54]. DINC supports a range of predefined modules, including topologies, solver, slicer, architectures, targets, and tester. Under topologies, DINC currently supports Fat-Tree, Folded-Clos, and BT ISP. The P4 slicers currently support manual configuration inputs, where inputs come from manual marking or are auto-generated by a data plane automation framework like Planter. DINC supports two architectures: v1model [35] and TNA [21] and coexists with simple forwarding, RARE [33], and Intel's switch.p4.

9 EVALUATION

Our evaluation focuses on three key questions: (i) does DINC enable deployment of large in-network computing algorithms that were not previously feasible (§ 9.2)? (ii) is DINC suitable for different network topologies with various configurations and scales (§ 9.3)? and (iii) is DINC's performance sensitive to network configurations and parameter tuning (§ 9.4-§ 9.5)?

9.1 Evaluation Setup and Datasets

9.1.1 Topologies. We evaluate DINC using two common network scenarios — a Folded-Clos based data center network topology (shown in Figure 11(a) with 3 core, 6 aggregation level, and 24 edge switches, connecting around ten thousand servers), and a large ISP backbone network - British Telecom (BT) (shown in Figure 11(b) with more than a thousand nodes: 8 inner core, 12 outer core, 63 metro, and 925 tier 1 switches). An aerial view of the BT topology is shown in Figure 18.

Using the two topologies, Table 2 shows a subset of potential ingress/egress switch setups. A total of five setups are presented, and are combinations of two dimensions: communication direction and deployment view. The direction of communication in a network may depend on a service requesting or responding, going between Core (C) and Edge (E) switches in Figure 11(a) or Inner Core (I) and Tier 1 (T) switches in Figure 11(b). Peer-to-peer communication between end servers is considered between Edge (E) switches or Tier 1 (T) switches.

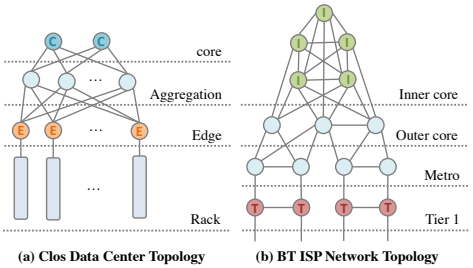


Fig. 11. Network topologies used for evaluation.

In terms of application deployment, a service may target all traffic going through a network, or a subset of users. In the first case, the network operator is the one that defines and controls the deployment. In the second, which fits e.g., campus networks, only a subset of the network might be used and have proprietary constraints.

Setup	Setup 1		Setup 2		Setup 3		Setup 4		Setup 5	
Topo.	Clos	BT	Clos	BT	Clos	BT	Clos	BT	Clos	BT
Ingress	All E	All T	All C	All I	One E	One T	All C	All I	One E	One T
Egress	All C	All I	All E	All T	All C	All I	One E	One T	One E	One T

Table 2. Network ingress & egress scenarios using the topologies shown in Figure 11.

9.1.2 Workloads. We use several popular in-network computing programs from different domains: in-network telemetry (INT) using PINT [2], load balancing using Pegasus [30], and in-network ML inference based on Planter [48, 57] with 5 ML models. These applications represent advanced in-network computing services, yet they require significant resources and need optimization when co-deployed with other data plane forwarding functions.

9.1.3 Metrics. The efficiency of distributed in-network computing is evaluated for the following: *1. Number of used nodes:* The number of nodes required to deploy an in-network computing program. This number depends on the total number of available nodes. *2. Hops per path:* The number of hops required to complete a given in-network computing program. Hop number is directly proportional to the latency, and hops are used to represent latency, and to ensure the evaluation is unbiased in terms of network setups and equipment selection. Cumulative distribution function (CDF) of hops provides further insights into the distribution of required hops per path. *3. Duplication of segments:* in topologies with multiple in-out paths, duplicate segments may exist on different paths. The amount of duplication is relative to the total number of nodes and shows the efficiency of the deployment strategy. *4. Execution time:* Job completion time of the ILP solver is critical to its application. The ILP solver's execution time depends on multiple factors, ranging from the topology to the program, and the solver will be generally scalable if the time increases in a linear manner.

9.1.4 Environment: Both large-scale simulation and small-scale hardware tests are conducted. The simulation is based on Mininet and BMv2, and hardware tests run on Tofino using APS-Networks BF6064X-T ($64\times100G$) and two NetBerg Aurora 710 ($32\times100G$) switches. Tofino compiler is further used for feasibility testing. Further details are in Appendix G.

9.2 Functionality

Simulation: DINC is evaluated both on the Folded-Clos data center network and BT topology, using the five setups and seven workloads. We measure the resource consumption and number of hops on both single and distributed deployments for all workloads, as summarized in Table 3.

DINC is capable of processing distributed deployment problems at data center level with about 10,000 servers or at ISP level with around 1000 switches. As Table 3 shows, all workloads, including

Program	Stand Alone			Segment	Setup	Folded-Clos Topology					BT ISP Topology				
	Stage	Alone	Coexist			Path	Dis.	Nodes	Hops	Dup.	Path	Dis.	Nodes	Hops	Dup.
INT-PINT [2]	7	✓	✗	5	4	18	✓	10/33	3/3	9	36	✓	3/1008	1.42/3.92	8
LB-Pegasus [30]	8	✓	✗	4	1	432	✓	30/33	2/3	56	26512	✓	400/1008	3.11/3.81	796
ML-NB [57]	8	✓	✗	2	4	18	✓	9/33	2/3	7	36	✓	6/1008	2.92/3.92	4
ML-SVM [56]	9	✓	✗	3	5	6	✓	8/33	3/3	5	12	✓	7/1008	3/5	4
ML-DT [48]	2	✓	✓	2	3	18	✓	1/33	1/3	0	36	✓	1/1008	1/3.92	0
ML-XGB [56]	6	✓	✗	4	2	432	✓	9/33	2/3	11	26512	✓	410/1008	2.76/3.81	422
ML-RF [56]	6	✓	✗	3	1	432	✓	30/33	2/3	33	26512	✓	400/1008	3.11/3.81	405

Table 3. Sample programs Supported by DINC. Segmentation (Seg.) details can be found in [54]. Setups refers to Table 2. Duplication (Dup.) - number of duplicated segments. ✓/✗ Deployment feasibility. Distribution (Dis.) - distribution feasibility. Nodes - nodes used/total. Hops - average used hops/path length.

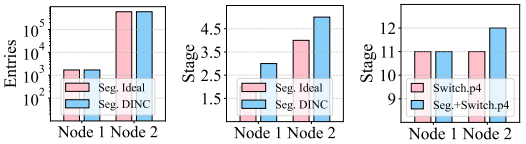
those that cannot coexist with RARE switch functionality (✗ in Coexist column) are feasible in a distributed deployment, coexisting with network functionalities (Dis. column). As some programs, e.g., ML-DT, coexist with other switch functions, DINC selects the best node along the path to optimally utilize resources and minimize latency. Such resource optimization advantages can be found in all distributed deployed programs.

In the comparison between standalone and distributed deployment, as illustrated in Table 3 (in the same row), consider the example of ML-RF/XGB. Under Setup 2, with a Folded-Clos data center configuration featuring 3 core, 6 aggregation, and 24 edge switches (equivalent to 1000 servers), the solver successfully resolves 4 segments. In this setup, more than 400 distinct data paths require service deployment. DINC's implemented algorithm deploys segments on only 9 out of 33 devices, achieving service completion with an average of 2 (out of 3) hops. Additionally, there are only 9 secondary segments duplicated, compared to deploying on each individual data path (which would require thousands of repeated segments) or alternatively routing all flows requiring service through a single path or a subset of paths. This significantly enhances deployment efficiency and reduces resource consumption.

In the comparison between distributed deployment under different path setups (among rows in Table 3), workloads in BT topology using all-to-all scenarios (setups 1 & 2) require using less than 40% of the nodes, and each path requires on average less than one duplicated segment. For one-to-one and one-to-all scenarios (setups 3-5), not all the devices are needed for complete task execution. In the data center setting, the number of all-to-all deployments requires one-third of nodes when data is arriving from the core (e.g., incoming to the data center). Tasks for outward flows may require all nodes if a latency optimization is applied (comparing ML-RF and ML-XGB). This shows that DINC's ILP solver can reach optimal planning with efficient node utilization and limited duplication under different topologies and traffic path conditions for both use cases.

DINC overheads: 1. *Communication Overhead.* Considering the scenarios listed in Table 3, the maximum DINC header size needed is 20B. This translates to 3.77% traffic overhead in the ML scenarios for packet classification. Using the ML models for flow-level classification using the first 30 packets per flow [7, 34], this overhead can be further reduced to 1.18%. For in-network telemetry application, specifically under the Hadoop workload [2, 38], the communication overhead is as low as 0.08%. 2. *Resource Overheads.* DINC's distributed deployment does not introduce additional memory overheads, as illustrated in Figure 12(a). However, DINC's distributed deployment does require additional pipeline stages. In comparison to a non-distributed program (the ideal case, which is calculated from the unsegmented program), the deployment of sliced segments requires 2 additional stages overhead in total, as shown in Figure 12b. The two nodes indicate the two consecutive connected network devices used for distributed deployment. These extra stages are primarily used to ensure the sequential execution of segments, as discussed in Section 6.2. Due to

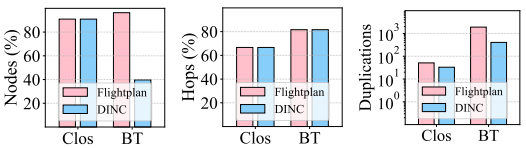
this design characteristic, other programs in Table 3 entail the same resource overhead. Based on the stage consumption results in Figure 12(b), Figure 12(c) presents the co-deployment together with the L2/L3 switch with 15 network features (switch.p4), which is an advanced version of the RARE router [33]. Compared to RARE in Figure 2, switch.p4 consumes more resources, which is used here for a stress test. The figure shows that a 3-stage segment in the first network device (Node 1) can be co-deployed without any stage overheads, while a 5-stage segment requires one extra stage overhead on the following Node 2.



(a) Memory (b) Stage (c) Stage (WS)

Fig. 12. Overhead of DINC on distributed deploying ML-RF [56] on two nodes, same as in the last row of Table 3. Seg. Ideal - resource consumption of segments calculated from the unsegmented program, WS - with switch functionality (switch.p4).

Comparison with state-of-the-art: Flightplan [43] is a state-of-the-art solution for P4 program disaggregation. We apply the Flightplan planner with objectives in terms of latency, throughput, and hops. We compare a deployment of ML-RF [56] on Clos (similar to the topology used in the Flightplan [43]) and BT topologies using both DINC and Flightplan. As Figure 13 shows, given the same program segmentation, in the Clos topology (33 nodes/switches), DINC and Flightplan use the same number of nodes and require the same number of hops for program completion (classification result). This is a result of a symmetric topology. Benefiting from the ILP planner, DINC requires 35.3% less segment duplications, saving 11.3% of memory resource. The benefits of DINC are demonstrated on the larger BT wide area network (1008 nodes), where DINC uses only 39.7% of the nodes, compared with Flightplan's 96.3%, and reduces the number of duplicated segments from 1901 to 405, saving 54.5% memory resources while maintaining the same number of hops. Compared to the heuristics used in Flightplan, DINC's ILP Planner is able to handle complex environments better and place segments efficiently.



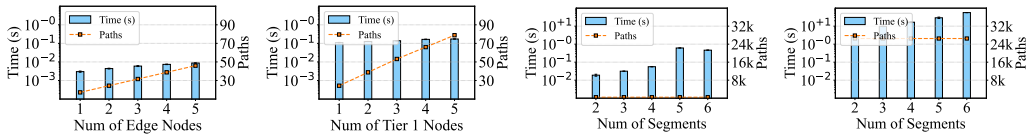
(a) Nodes. (b) Hops. (c) Dup.

Fig. 13. Resource consumption comparison between Flightplan [43] and DINC on deploying ML-RF [56] on both BT ISP and Folded-Clos topology with setup 1, same as in the last row of Table 3.

Hardware test: We evaluate DINC on a small-scale hardware setup using the distributed RF-ML program on Point-to-Point and Fat-Tree topologies constructed using three Tofino switches, covering the majority of scenarios in Table 2 & Table 3. RF-ML is chosen as Planter provides a functionality validation test. The result shows the model functions correctly with the same accuracy as a standalone deployment. For detailed setup see Appendix G.

9.3 Scalability

Using ML-RF as a leading example, we explore the scalability of the DINC solver on both two topologies in terms of two key factors: the number of paths of the topology and the number of segmentations of the in-network computing program. Despite the exponential increase in solving time for complex topologies, our tests on medium and large networks show that DINC solver efficiently handles the problem within a reasonable timeframe. It achieves millisecond-level planning for the core to single switch deployment (setup 4) for both Folded Clos data center topology (which is able to support around 10,000 servers) and BT ISP topology (for the whole UK). Network-level deployment for a 6-segment ML-RF program across all cores and edge switches is solved in 1 second for the Folded-Clos topology (400 paths) and 1 minute for the BT ISP topology (26k paths). Looking into the trend of time consumption, as reported in Figure 14(a) and (b), a setup where input is all Edge/Tier 1 switches and output is Edge switches. When the number of Edge/tier1



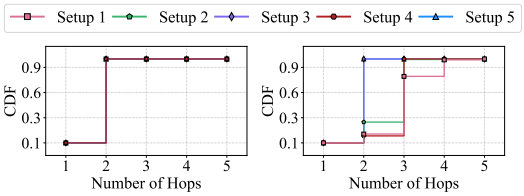
(a) Folded-Clos Topo. (b) BT ISP Topology. (c) Folded-Clos Topo. (d) BT ISP Topology.

Fig. 14. DINC ILP solver runtime and number of *In-Out Paths* scaling with coverage area (the number of edge/tier 1 switches) or number of program segments.

nodes increases, the number of paths will also increase linearly, while the time required by the DINC solver increases but slows down as the number of paths increases. When we test the number of segments, we apply setup 4 for chasing a relatively large usage of paths. In Figure 14(c) and (d), when the number of segments of in-network computing programs increases, the time consumption increases exponentially. However, this is acceptable due to the number of segmentations is usually bounded by a small number.

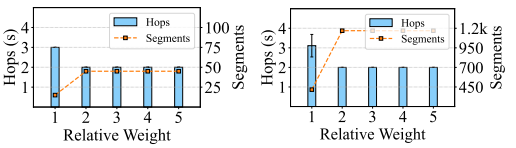
9.4 Performance

We also use ML-RF as a leading example to show how DINC performs under different network scales. The ML-RF program requires 6 stages and is able to be deployed within a single path with consecutive switches. Figures 15 show a hops CDF along all 5 sets of in-out paths generated by 5 different setups. As reported from Figure 15(a), no matter which setup on Folded-Clos topology and due to its relatively simple and strongly symmetric topology, ML-RF can be executed within two hops on all paths.



(a) Folded-Clos Topo. (b) BT ISP Topology.

Fig. 15. CDF of hops needed to complete a program. Setups are in Table 2 and Figure 11.



(a) Folded-Clos Topo. (b) BT ISP Topology.

Fig. 16. How relative objective weight of w (Equation 7) can influence the DINC deployment strategy.

In Figure 15(b), for BT ISP topology, setups 3 and 5 exhibit the best performance in terms of latency as they process the traffic from the user level (input only comes from one switch). Under this circumstance, there is no conflict among the objectives of lower latency, memory, and the number of segmentations, allowing for closer-to-source deployment. Setup 2 and 4 have a similar performance and share a relatively larger number of hops than the optimized solution. This is because their tasks come from a deeper network and the input switches are inner core switches. These setups contain more in-out paths but the traffic comes in at the places where each path is easier to share the usage of segmentation. Setup 1 has the worst latency performance and it requires 4 hops to ensure all paths finish the processing of the program. It is because setup 1 represents the deployment of services from all tier 1 edge switches to the inner core. The shared switches, usually at the inner part of the network, are at the end of each route. It will be an extreme waste of resources if we deploy duplicated segments on all paths for lower latency. For this reason, even though both setups 1 & 2 have the same number of paths and just swap input and output switches, they have slightly different CDF results.

Therefore, we observe that DINC performs effectively in the distributed deployment of programs, whether in common data center networks composed of 33 network devices or in large-scale ISP networks with up to 1008 network devices.

9.5 Sensitivity

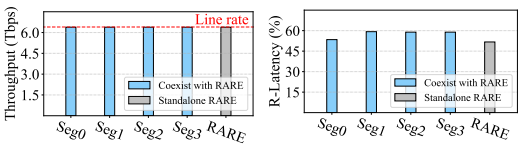
We test the sensitivity and the trade-offs of the DINC solver to the average number of hops and the required number of duplicated segmentations, as the relative weights between latency and other objectives are changed. Figure 16 shows that for both two topologies when we increase the relative weight of latency above a threshold, the average number of hops will decrease. Yet, to ensure functionality, the provided strategy requires more duplicate segments. This implies that DINC users have the flexibility to adjust deployment strategies based on the defined objective function and its associated weights. We note that the DINC is relatively robust and the deployment strategy will not change abruptly when relative weights change.

The robustness of DINC's distribution planner is further explored for ML-RF. When changing the number of segments from 2 to 6, DINC consistently generates identical distribution results.

9.6 Throughput & Latency

The part shows the evaluation of the throughput of different decomposed RF model segments on an Intel Tofino switch $64 \times 100G$, coexisting with the RARE router [33]. The latency part shows the relative latency of each segment+RARE with *switch.p4* (an L2/L3 reference switch). Appendix G provides details of the setup.

For the sample distributed RF segments, As shown in Figure 17, all segments are able to coexist with the RARE router program and are able to achieve a full line rate of 6.4Tbps bit rate. For the latency of each segment, the coexistence of the in-network computing application segment will increase the 10% of the relative clock cycle. However, even coexists with RARE, all deployed segments have a relatively low latency, which is lower than 60% of the reference design (switch.p4). The full line rate in the result proved that the DINC-added bitmap and metadata passing mechanism will not limit the throughput of the system. The relative latency shows that the coexistence of network functions by using DINC will not significantly increase the consumption of the clock cycle.



(a) Throughput (Tbps). (b) R-Latency.

Fig. 17. Throughput and Latency on Hardware.

10 DISCUSSION AND LIMITATIONS

DINC is an important step toward distributed in-network computing and improved utilization of data plane resources.

Scope of DINC. Not all in-network computing programs are suitable for a distributed deployment. DINC is well-suited for stateless programs without packet recirculation. In stateful programs, the limitation stems from potential multi-path routing, rather than DINC's distributed deployment. Packets going through different paths will experience different states (e.g., counter value). This can be resolved by setting DINC to generate single-path results. In programs requiring recirculation, DINC cannot guarantee that the segmented program can resend packets to the switch of the first deployed segment. Instead, recirculation can be transformed to longer programs (as in loop unrolling) before applying DINC.

DINC prerequisite. To run DINC, users need to provide a program, the program's segmentation, and resource requirements for each segment, and it is assumed users have a certain level of understanding of their programs. Automation of the segmentation and resource tasks can be achieved by adding DINC new modules (e.g., [27]). Additionally, the Planner requires the network's topology, routing paths, and available resources at each node. The architecture and base program running on each network device are needed to auto-generate the distributed code.

Stage sharing. Although switch functionality consumes considerable resources in pipeline, it is still possible to share stages with an in-network application. Currently, DINC does not exhaust these spaces during the planning process and leaves them as elastic "spaces". They are sometimes used for operations like assigning metadata to header and bitmap checking.
Slicing applications in other languages. In general, DINC is designed for P4 code decomposition and distribution on programmable network devices. To support other languages, e.g. NPL or Domino [41], the P4 Generator of DINC should be replaced with the relevant language' generator.
Partitioning characteristics. Different program partitioning can impact usability. Finer-grain segmentation yields better results from the planner. However, increasing segmentation also increases the computational burden on the planner. Additionally, the placement of slices can impact the amount of metadata used, thereby influencing communication overheads. This consideration can be incorporated into optimization constraints and objective functions.
Runtime updates. The control plane can drive runtime control and updates based on the planner's results. However, when adding or altering services, or when there are changes to network topology, it is necessary to rerun DINC and update affected nodes. For certain modifications, such as adding new routing paths, it is not sufficient to run DINC on the added routing paths rather than the entire topology. While this incremental approach may not yield the optimal result, it avoids non-critical updates and can significantly reduce computational overhead. However, to find the global optimal result, we have to run DINC over the entire topology.
Failed DINC compilation. Three reasons can lead to a compilation failure. 1. *Insufficient Resources for Deployment*: If a path contains less resources than the minimum needs of a given program, or if the network resources cannot jointly satisfy the minimal deployment requirement across multiple paths, no valid planning solution exists. 2. *Inaccurate resource estimation*: An overestimation of device resources by the central controller, an underestimation of required resources by the user, or an underestimation of DINC's code merging overheads can lead to a failed compilation. As these issues are reported during compilation, they can be fixed by adjusting the corresponding resource constraints and re-running DINC. 3. *Inadequate Code Slicing*: In certain instances, excessively coarse-grain code partitioning can result in deployment failures. Very large code blocks may exceed a single device's resources, requiring a finer-grain partitioning, followed by a re-execution of DINC.
Network failures. A device failure does not necessarily lead to a program failure. If multiple paths exist between the source and destination (prior to the failure), operations continue without disruption. This is due to the handling of link failures by the network's inherent load balancing and routing mechanisms, underscoring DINC's advantage of preserving the original routing and network functionalities. In the case of a device failing along a single route, a new route needs to be found and DINC should be incrementally run for this route.
Stages consumption In the evaluation, we use models and configurations that are sometimes intentionally larger than in the original publication. For example, for ML-RF [57], it is possible to deploy a small 3-stage model, with a reduced level of accuracy, while we use a larger model that consumes 6 stages. This is as one of the DINC's aims is to improve the scalability of in-network programs, and ML model sizes in particular.
Planning objectives and constraints. The constraints and objectives above are provided as an example and are suitable for most in-network computing workloads. More objectives, such as latency or throughput constraints, variations of use-case and resources constraints, and minimizing packet overheads [9] can be specified in a planner module (§ 7). DINC also supports heterogeneous devices by adjusting constraints on the corresponding resources.
DINC Applications. DINC is designed to execute a single large in-network computing program across multiple network devices in a distributed manner. It supports various in-network computing applications, such as telemetry [2], aggregation [28], inference [58], and load balancing [30].

Furthermore, by relaxing planner constraints, DINC can potentially offer distributed deployment strategies for a broader range of computing tasks (e.g., [14, 15, 31]).

11 RELATED WORK

NFV service chain planning. Service distribution is common in virtual network functions (VNFs) [10, 11, 22–24, 49]. Network function virtualization (NFV)-based work mainly focused on traditional processing, where the network serves just as a medium, and latency and throughput are the main objectives. In DINC, the network is both the processing element and the medium, and in-network services are deployed across many paths. DINC's deployment augments network functionality without changes to routing.

Distributed in-network computing. Previous research efforts [8, 9, 32] primarily focused on distributing programs across devices to support a single path through the network. Notably, Hermes [9] and SPEED [8] did not support slicing of a given program. In SRA [32], Match-Action tables were disaggregated, and routing rules were introduced to reach the next table. None of these studies addressed the many-paths, any-to-any routing problem. ClickINC [50] required program translation into a new language and only supported symmetric topologies. Flightplan [43] demonstrated P4 program disaggregation, however, its BSP planner is less suitable for large-scale complex topologies. Flightplan also introduces changes to routing rules and does not explore coexistence with network functionality. In contrast, DINC does not modify routing rules and scales to large networks.

Work	Slice Code	ILP Planner	Any-to-Any	Code Gen.
Hermes [9]	✗	✓	✗	✗
SRA [32]	✓	✗	✗	✓
SPEED [8]	✗	✓	✗	✗
ClickINC [50]	✗	✗	Partial	✓
Flightplan [43]	✓	✗	Partial	✓
DINC	✓	✓	✓	✓

Table 4. Comparison between related works. Flightplan supports any to any while changing routing rules.

12 CONCLUSION

This work presented DINC, a distributed in-network computing framework, decomposing in-network computing programs and generating planning strategies to distribute decomposed segments onto devices within the network. Experimental results show that large-size in-network computing programs can be deployed within the network using efficient decomposing and optimal planning. DINC boosts applications' performance by utilizing network resources without compromising functionality.

13 ACKNOWLEDGMENTS

This paper complies with all applicable ethical standards of the authors' home institution. This work was partly funded by VMware, EU Horizon SMARTEDGE (101092908, UKRI 10056403), ARO W911NF-23-1-0088 and W911NF-23-1-0064. We acknowledge support from Intel. For the purpose of Open Access, the author has applied a CC-BY public copyright license to any Author Accepted Manuscript (AAM) version arising from this submission.

REFERENCES

[1] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (Virtual Event, USA) *(SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 13–26. https://doi.org/10.1145/3482898.3483365

[2] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 662–680.

[3] Kenneth P Birman. 1993. The Process Group Approach to Reliable Distributed Computing. *Commun. ACM* 36, 12 (1993), 37–53.

[4] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. 2021. Switches for HIRE: Resource Scheduling for Data Center In-Network Computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 268–285.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[6] BT. 2022. BT Group plc Annual Report 2022. https://www.bt.com/bt-plc/assets/documents/investors/financial-reporting-and-news/annual-reports/2022/2022-bt-annual-report.pdf [Online, accessed June 27, 2023].

[7] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pForest: In-Network Inference with Random Forests. *CoRR* abs/1909.05680 (2019). arXiv:1909.05680 http://arxiv.org/abs/1909.05680

[8] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. 2020. SPEED: Resource-Efficient and High-Performance Deployment for Data Plane Programs. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

[9] Xiang Chen, Hongyan Liu, Qingjiang Xiao, Kaiwei Guo, Tingxin Sun, Xiang Ling, Xuan Liu, Qun Huang, Dong Zhang, Haifeng Zhou, et al. 2022. Toward Low-Overhead Inter-Switch Coordination in Network-Wide Data Plane Program Deployment. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 370–380.

[10] Yang Chen. 2020. *Network Update and Service Chain Management in Software Defined Networks*. Temple University.

[11] Yang Chen and Jie Wu. 2018. NFV Middlebox Placement with Balanced Set-up Cost and Bandwidth Consumption. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[12] Ryan A Cooke and Suhaib A Fahmy. 2020. A Model for Distributed In-Network and Near-Edge Computing with Heterogeneous Hardware. *Future Generation Computer Systems* 105 (2020), 395–409.

[13] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1726–1738.

[14] Joaquín Delgado Fernández, Sergio Potenciano Menci, Chul Min Lee, Alexander Rieger, and Gilbert Fridgen. 2022. Privacy-Preserving Federated Learning for Residential Short-Term Load Forecasting. *Applied energy* 326 (2022), 119915.

[15] Alex Galakatos, Andrew Crotty, and Tim Kraska. 2018. *Distributed Machine Learning*. Springer New York, New York, NY, 1196–1201. https://doi.org/10.1007/978-1-4614-8265-9_80647

[16] Vladimir Gurevich and Andy Fingerhut. 2021. P4-16 Programming for Intel Tofino Using Intel P4 Studio. In *2021 P4 Workshop*.

[17] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[18] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. 2023. LOBIN: In-Network Machine Learning for Limit Order Books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 159–166.

[19] C-L Hwang and Abu Syed Md Masud. 2012. *Multiple Objective Decision Making — Methods and Applications: A State-of-the-Art Survey*. Vol. 164. Springer Science & Business Media.

[20] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4→NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–9.

[21] Intel. 2021. P4_16 Intel® Tofino™ Native Architecture – Public Version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf

[22] Maryam Jalalitabar, Evrim Guler, Danyang Zheng, Guangchun Luo, Ling Tian, and Xiaojun Cao. 2018. Embedding Dependence-Aware Service Function Chains. *Journal of Optical Communications and Networking* 10, 8 (2018), C64–C74.

[23] Maryam Jalalitabar, Yang Wang, and Xiaojun Cao. 2019. Branching-Aware Service Function Placement and Routing in Network Function Virtualization. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 1–6.

[24] Yu Jiang, Xiaolong Xu, Kunda Lin, and Weihua Duan. 2021. TSC-ECFA: A Trusted Service Composition Scheme for Edge Cloud. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 58–65.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[26] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. 2022. NetREC: Network-wide in-network REal-value Computation. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE, 189–197.

[27] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. 2020. Tracking P4 Program Execution in the Data Plane. In *Proceedings of the Symposium on SDN Research*. 117–122.

[28] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.

[29] Jong-Hyouk Lee and Kamal Singh. 2020. SwitchTree: In-network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications* (2020), 1–12.

[30] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with in-Network Coherence Directories. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 22, 20 pages.

[31] Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. 2020. Concerto: Cooperative Network-wide Telemetry with Controllable Error Rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 114–121.

[32] Hongyan Liu, Xiang Chen, Qun Huang, Haifeng Zhou, Dong Zhang, and Chunming Wu. 2020. SRA: Switch Resource Aggregation for Application Offloading in Programmable Networks. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 1–6.

[33] Fréderic Loui, Csaba Mate, Alexander Gall, and Maxime Wisslé. 2022. Project Overview: Router for Academia Research & Education (RARE). (2022).

[34] Nour Moustafa and Jill Slay. 2015. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set). In *2015 military communications and information systems conference (MilCIS)*. IEEE, 1–6.

[35] Barefoot Networks. 2021. Version 1.0 Switch Architecture Model. Website. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4

[36] Manfred Padberg and Giovanni Rinaldi. 1991. A Branch-and-cut Algorithm for the Resolution of Large-scale Symmetric Traveling Salesman Problems. *SIAM review* 33, 1 (1991), 60–100.

[37] R Pant, T Russell, C Zorn, E Oughton, and JW Hall. 2020. Resilience Study Research for NIC–Systems Analysis of Interdependent Network Vulnerabilities.

[38] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside The Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.

[39] SamKnows. 2023. 21CN Overview - More Details About BT Wholesale 21CN. https://availability.samknows.com/broadband/exchanges/21cn_overview

[40] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI*. 785–808.

[41] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

[42] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 518–532.

[43] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 571–592.

[44] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing on Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[45] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. 2020. ZipLine: In-Network Compression at Line Speed. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 399–405. https://doi.org/10.1145/3386367.3431302

[46] Shuhe Wang, Zili Meng, Chen Sun, Minhu Wang, Mingwei Xu, Jun Bi, Tong Yang, Qun Huang, and Hongxin Hu. 2020. SmartChain: Enabling High-Performance Service Chain Partition between SmartNIC and CPU. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.

[47] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He, and Kavé Salamatian. 2016. Transparent Flow Migration for NFV. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. 1–10. https://doi.org/10.1109/ICNP.2016.7784446

[48] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33.

[49] Qi Xu, Deyun Gao, Taixin Li, and Hongke Zhang. 2018. Low Latency Security Function Chain Embedding Across Multiple Domains. *IEEE Access* 6 (2018), 14474–14484.

[50] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. 2023. ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 798–815.

[51] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. 2023. Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways. *IEEE Internet of Things Journal* (2023).

[52] Mingyuan Zang, Changgang Zheng, Tomasz Koziak, Noa Zilberman, and Lars Dittmann. 2023. Federated learning-based in-network traffic analysis on IoT edge. (2023).

[53] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. 2023. QCMP: Load Balancing via In-Network Reinforcement Learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*. 35–40.

[54] Changgang Zheng, Haoyou Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. 2023. DINC - GitHub Repository. https://github.com/In-Network-Machine-Learning/DINC

[55] Changgang Zheng, Haoyou Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. 2023. DINC Artifact. https://doi.org/10.5281/zenodo.10034834

[56] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. IIsy: Practical In-Network Classification. https://doi.org/10.48550/ARXIV.2205.08243

[57] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating In-Network Machine Learning. https://doi.org/10.48550/ARXIV.2205.08824

[58] Changgang Zheng and Noa Zilberman. 2021. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 12–14.

[59] Yu Zhou, Jun Bi, Cheng Zhang, Bingyang Liu, Zhaogeng Li, Yangyang Wang, and Mingli Yu. 2019. P4DB: On-the-fly Debugging for Programmable Data Planes. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1714–1727.

A LIST OF ACRONYMS

The acronyms used in this paper are listed in Table 5.

B TERMS AND DEFINITIONS

The variables defined in this paper and their meaning are listed below. These variables are mainly used in Section 5. The terms used are the following:

- $\mathcal{D}$: Deployable devices.
- L_d^e: The execution latency of element e on device d.
- N_d: Number of deployable devices.
- N_p: Number of routing paths links input/output switches.
- N_r: Number of resources in the device.
- $\mathcal{P}$: Routing paths.
- p_i: The i th routing path.
- R_d^r: The available type r resource on device d.
- $X_{e\to d}$: Deployment decision/strategy.

Acronyms	Definition
ALU	Arithmetic Logic Unit
DT	Decision Tree
ILP	Integer Linear Programming
INT	In-Network Telemetry
LB	Load Balancing
ML	Machine Learning
NB	Naïve Bayes
PISA	Protocol Independent Switch Architecture
P4	Protocol-Independent Packet Processing
RF	Random Forest
SVM	Support Vector Machine
TNA	Tofino Native Architecture
XGB	Extreme Gradient Boosting (XGBoost)

Table 5. Acronyms.

C ILP SOLVING DETAILS

Recall that $X_{e\to d}$ are binary decision variables. Minimize the scalarized objective function (7) under constraints (2), (3) and (4) can be rewritten as the following integer linear programming problem (ILP):

$$\min_{X_{e\to d}} w_R \sum_{r=1}^{N_r}\sum_{d=1}^{N_d}\sum_{e=1}^{N_e} O_e^r X_{e\to d} + w_L \sum_{i=1}^{N_p}\sum_{j=1}^{l_i}\sum_{e=1}^{N_e} L_e^{p_i^j} X_{e\to p_i^j}, \tag{8a}$$

$$\text{s.t.,} X_{e+1\to p_i^j} - \sum_{k=1}^{j-1} X_{e\to p_i^k} \le 0, \forall e \in [N_e - 1], i \in [N_p], j \in [l_i], \tag{8b}$$

$$-\sum_{k=1}^{l_i} X_{e\to p_i^k} \le -1, \forall i \in [N_p], \tag{8c}$$

$$\sum_{e=1}^{N_e} O_e{}^r X_{e\to d} \le R_d^r . \forall d \in [N_d], r \in [N_r], \tag{8d}$$

$$X_{e\to d} \in \{0,1\}, \forall e \in [N_e], d \in [N_d]. \tag{8e}$$

Searching for the optimum solution of an ILP is NP-hard. We tackle this problem using the branch-and-cut method [36] that combines the cutting plane method and brand-and-bound method. Let $\vec{X} := \{X_{e\to d}\}_{e,d}$ be the vector of the decision variables. The first step is to relax the integer constraint (8e) into

$$0 \le X_{e\to d} \le 1, \forall e \in [N_e], d \in [N_d]. \tag{9}$$

rewrite the objective function (8a) and constraints (8b)-(8d) in the standard form of

$$\min_{\vec{X}} \vec{c}^T \vec{X}, \text{s.t., } \mathbf{A}\vec{X} = \vec{b}. \tag{10}$$

After checking the feasibility of problem (10), we set $o^\star = \infty$, put the original LP problem (10) into set $\mathcal{L}$ and iterates the following steps until set $\mathcal{L}$ is empty:

(1) Pop a linear programming problem $\mathcal{P}$: $\min_{\vec{c}^T\vec{X}}$, s.t., $A^{(p)}\vec{X} = \vec{b}^{(p)}$ from set $\mathcal{L}$.
(2) Solve problem $\mathcal{P}$ using simplex method. Let $\vec{X}^{(p)}$ and $o^{(p)} = \vec{c}^T\vec{X}^{(p)}$ be the objective vector and value to problem $\mathcal{P}$.
(3a) Check if each element $\vec{X}^{(p)}$ belongs to $\{0, 1\}$.
- If $X_{e\to d} \in \{0, 1\}, \forall x \in \vec{X}$, indicating the optimum solution of problem $\mathcal{P}$ is an 0/1 integer. Therefore if $o^\star > o^{(p)}$, we can improve our strategy by selecting $\vec{X}^\star = \vec{X}^{(p)}$ and set $o^\star = o^{(p)}$.
- If there exists $X_{e\to d} \notin \{0, 1\}$ and the optimum solution satisfies $o^\star \geq o^{(p)}$, there may exists a better solution. We then push two new LP problems $\mathcal{P}_1$ and $\mathcal{P}_2$, where problem $\mathcal{P}_1$ is the original LP problem $\mathcal{P}$ with an additional constraint that $X_{e\to d} = 0$, and problem $\mathcal{P}_1$ is problem $\mathcal{P}$ with additional constraint that $X_{e\to d} = 1$.

The iteration continues until the LP set $\mathcal{L}$ is empty. If $o^\star = \infty$ after the iteration finishes, then the integer programming problem is infeasible. Otherwise, $\vec{X}$ is the optimal Pareto operation point we are searching for.

D SLICER DETAILS

Multiple types of information among markers are used to indicate the DINC slicer the segment identifier and the position. For example, in Table 1 lines 13 and 15, the Slice identify the segment ID is 0 and the End shows the end of this segment. Position in markers shows the position of that segment is control apply (line 13), while the segment in control block is marked as control (line 2). All segments with the same ID and position will be merged and saved for future use.

The dependency information is embedded in the marker Previous (Table 1 line 13, 16, and 19). It indicates if this segment has any prerequisite segments. The prerequisite segments can be none, single, or multiple.

The required resource information, metadata in/out, and any other required information are used in a similar way as a dependency to be added to the marker. It will be totally flexible for the DINC slicer to add or remove required resources, if the current resource marker is missing for a specific segment, it will be marked as zero. The resource with the same segment ID will be added together without being influenced by the marker position.

This is not the only option for slicer design and the DINC's modular framework allows personalized slicer design that is different from the current design (§ 7.3).

E RUNTIME FAIRNESS

We discuss the runtime fairness of DINC's deployment of algorithms from two perspectives. 1. *Fairness in terms of planning*: Network infrastructure owners (or service providers) have the flexibility to ensure fairness in the deployment through various strategies, such as setting constraints on the maximum resource usage per device per resource category or imposing constraints on the overall resource usage. Alternatively, they may opt for more aggressive strategies, such as a first-come, first-served approach. The choice of strategy depends on the characteristics and requirements of the network operator. 2. *Fairness in terms of services*: DINC, particularly in scenarios devoid of intricate operations like multicast or recirculate, inherently provides a level of fairness in network services. This is because DINC does not disrupt the load balancing and forwarding of the regular network, which is one of its distinguishing features. If absolute fairness is required, maximum throughput limits for individual services can be configured on switches. Nonetheless, services

that may introduce additional traffic require the addition of constraints in the planner to mitigate potential impacts.

F BT TOPOLOGY DETAILS

BT is the UK's largest ISP, with a network spanning across the country. BT provides connectivity to 45% UK residential households and 1.2M business customers, as of 2022 [6]. BT's backbone topology is extracted from the real network topology of BT Wholesale 21CN [37, 39]. The BT Wholesale 21CN includes 5600 bottoms (MSAN) level exchanges, and 106 Metro nodes that are dotted around the country (typically in major cities). Beyond that, there are about 1100 Tier 1 MSANs and 4400 Tier 2 and 3 MSANs (only tier 1 nodes are directly connected to the Metro network), and 20 Core nodes (8 inner-core and 12 outer-core nodes) to handle very high-speed switching and routing between Metro nodes with up to 100Gbps capacity. The Tier 2 and Tier 3 nodes are managed by Openreach, a subsidiary of BT, and as they are separately administered, they are excluded from our evaluation. The spatial layout of these nodes in BT ISP topology is shown in Figure 18.

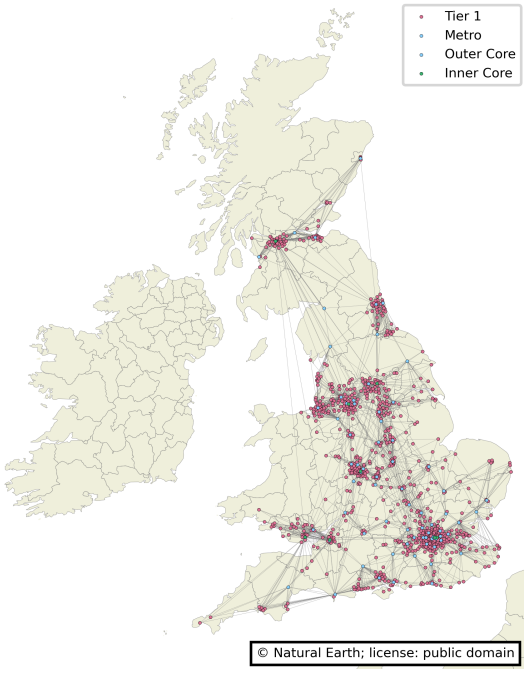


Fig. 18. An aerial view of the BT ISP topology used in the evaluation.

G ARTIFACT DESCRIPTION AND EVALUATION SETUP

DINC is an open-source framework. The code for the framework and its evaluation environment are available on GitHub [54] and [55]. A detailed step-by-step guide for using the framework is provided in `README.md` and `./src/help`. The DINC framework's *README.md* file provides detailed information on supported architectures, targets, use cases, solvers, and topologies. It also covers more advanced topics adding new architectures, targets, use cases, solvers, and topology modules.

The following setup was used to run the DINC framework and emulation on a server:

- Linux server runs Ubuntu 18.04.1 with installed software development environment SDE 9.9.0 (Tofino) or BMv2 1.15.0. Running the framework requires Python version 3.10 (or above) with several pre-installed packages.
- The command *python3 DINC.py -m -d <P4 program directory>* is used to run the framework. The running example is provided in [54].

For the performance evaluation on Tofino, the test setup includes 3 steps: 1. setup switch environment, 2. setup server environment, and 3. configure and run programs on the switch.

- The system test environment uses two types of switches. The first is APS-Networks BF6064X, an Intel Barefoot Tofino platform with 64×100G ports. The switch runs Ubuntu 18.04.1 and Barefoot's SDE 9.6.0 is used on the switch. The other is NetBerg Aurora 710, an Intel Barefoot Tofino platform with 32×100G ports. The switch runs 4.19.81 OpenNetwork Linux and Barefoot's SDE 9.9.0 is used on the switch.
- ESC4000A-E10 servers using AMD EPYC 7302P CPUs with 256GB RAM, Ubuntu 20.04LTS, and equipped with Mellanox ConnectX-5 100G NICs are used to send traffic to the switch using DPDK 20.11.1 and PktGen 21.03.0. Four CPU cores are dedicated per port.

- To test full throughput, a snake configuration is used, where traffic is looped from each port to the following one, enabling traffic across all 64 ports, which is a common practice in existing work (e.g., [13], [2]). Python scripts are used to generate and capture traffic. Simple forwarding achieves the baseline 6.4Tbps on the switch.
- To test the hardware functionality, switches are connected in both tree and point-to-point topology (BF6064 as the core and Aurora 710 as the edge), where traffic comes from the port on BF6064 to the following Aurora 710. Both switches run the data plane program distributed by DINC. Python scripts are used to generate and capture traffic for functionality verification.

H STATE-OF-ART COMPARISON - EXTENSION

Compared to distributed computing [3] and network service chains [47], distributed in-network computing faces unique challenges in minimizing its impact on existing networks. Unlike network service chains, which may affect routing rules, distributed in-network computing aims at a type of deployment which is transparent to other network functions. This assumption of no routing changes is common in many previous in-network computing works (e.g., [13, 25, 44]), in order to avoid additional network load or load imbalance and reduce the complexity of managing the network. Unlike computing-based scheduling, distributed in-network computing deployment lacks the resource abundance and deployment flexibility. One cannot simply issue a task to run on a certain network node, the task needs to traverse several other nodes to reach the processing node, thereby inadvertly employing multiple nodes in the processing task. This demands a deployment of a fully-functional in-network computing algorithm along each potential path that a packet may traverse. Similar to computing, where an operating system runs on every machine, distributed in-network computing needs to support normal network functionality on every network device, before other services can be deployed.

As Table 4 shows, prior research efforts like Hermes [9], SRA [32], and SPEED [8] do not fully facilitate “any-to-any“ deployment, resulting in additional routing overheads and the potential for network load imbalance. ClickINC [50] supports multi-path deployment, but it is centered on data center deployments and it makes assumptions on the network, such as symmetry, for its optimization. Flightplan [43] also focused on data centers, especially on support for rack-scale deployments where the scale is known. Moreover, Flightplan uses changes to the routing to achieve its processing objective. For example, Flightplan [43] Figure 7 shows a sample deployment where the firewall is only on p0e1 and p1e1. If traffic from p0h0 needs to go to p1h0 along the shown route, it will not traverse any firewall. To go through a firewall, routing rules need to be changed on p0a0 or p1a0, and a packet will need to traverse one of these switches twice.

In our comparison in §9.2, we employed the heuristic planner in Flightplan based on the optimization objective of minimizing latency increase for planning (as in the Flightplan artifact). Compared to DINC’s multi-objective ILP planner, the single-objective heuristic search exhibited limited scalability and lacked the trade-offs between objectives. However, it is worth noting that the performance of Flightplan and DINC may be changed or improved with the change of objectives, meaning the redesign planners. Currently, Flightplan’s planner accepts heuristics to prune the search since the topology can make it expensive, leading to DINC’s better performance on the larger BT topology. Flightplan also did not explore fairness, the closest constraints used were to avoid overallocation.