2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00043

Toward Low-Overhead Inter-Switch Coordination in Network-Wide Data Plane Program Deployment

Xiang Chen[1,2,3], Hongyan Liu[1], Qingjiang Xiao[3], Kaiwei Guo[3], Tingxin Sun[3], Xiang Ling[4], Xuan Liu[5,6], Qun Huang[2], Dong Zhang[3], Haifeng Zhou[1], Fan Zhang[1], Chunming Wu[1]
[1]Zhejiang University [2]Peking University [3]Fuzhou University
[4]Institute of Software, Chinese Academy of Sciences [5]Yangzhou University [6]Southeast University

***Abstract*—In modern networks, administrators realize their desired functions such as network measurement in several data plane programs. They often employ the *network-wide program deployment* paradigm that decomposes input programs into match-action tables (MATs) while deploying each MAT on a specific programmable switch. Since MATs may be deployed on different switches, existing solutions propose the inter-switch coordination that uses the per-packet header space to deliver crucial packet processing information among switches. However, such coordination introduces non-trivial *per-packet byte overhead*, leading to significant end-to-end network performance degradation. In this paper, we propose Hermes, a program deployment framework that aims to minimize the per-packet byte overhead. The key idea of Hermes is to formulate the network-wide program deployment as a mixed-integer linear programming (MILP) problem with the objective of minimizing the per-packet byte overhead. In view of the NP hardness of the MILP problem, Hermes further offers a greedy-based heuristic that solves the problem in a near-optimal and timely manner. We have implemented Hermes on Tofino-based switches. Our experiments show that compared to existing frameworks, Hermes decreases the per-packet byte overhead by 156 bytes while preserving end-to-end performance in terms of flow completion time and goodput.**

I. Introduction

Modern production networks such as data center networks (DCNs) [1, 2] have evolved from relatively static networks to highly programmable ones enabling dynamic reconfiguration. In these networks, administrators describe their desired packet processing logic in *data plane programs*. They compile these programs and install the output configurations on programmable switches, i.e., *program deployment*. In particular, given the resource limitations of a single programmable switch [3, 4], the deployment may be failed when input programs require more resources than the capacity of a single switch.

To this end, recent solutions propose *network-wide program deployment* [5, 6, 7]. They first decompose input programs into several *match-action tables* (MATs), which are the basic packet processing components that match packets with user rules and perform corresponding actions on packets based on matching results. Next, these solutions place each MAT on a specific programmable switch. They coordinate the MATs running on different switches to jointly process packets. By that means, they faithfully preserve switch resource limitations as well as the entire packet processing logic.

However, such inter-switch coordination brings non-trivial *per-packet byte overheads*. Specifically, MATs have *execution dependencies* on each other [8]. For example, in Figure 1(a),

Qun Huang, Chunming Wu, and Xuan Liu are corresponding authors.

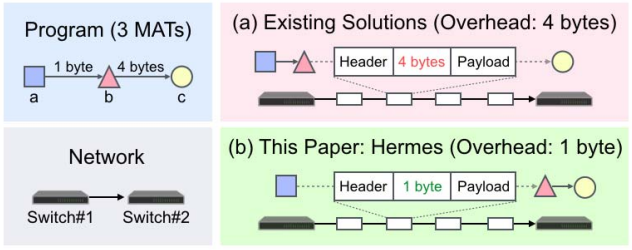


Fig. 1: *Example of the inter-switch coordination. The MAT a delivers 1-byte metadata to b while b delivers 4 bytes to c. We assume that each switch can only tolerate two MATs. (a) The program deployment made by existing solutions requires to piggyback 4-byte metadata on each packet. (b) Hermes reduces the per-packet byte overhead from 4 bytes to 1 byte by deploying b and c on the same switch.*

we assume that the MAT b uses indexes to manipulate an array of switch counters. Thus, it should be invoked after the MAT a that calculates indexes. If a and b are deployed on different switches, their dependency should be preserved to maintain the correctness of packet processing. Thus, existing solutions *piggyback* the processing results (e.g., indexes), i.e., *metadata*, of the upstream MATs (e.g., a in the above example) on the header space of each packet. When the current switch finishes its processing, it routes the packet carrying metadata to the next switch, in which downstream MATs (e.g., b) extract metadata from the packet headers and perform their subsequent processing. Thus, MAT dependencies are well preserved. However, a significant portion of the limited header space of each packet is dedicated to storing metadata. As a result, fewer bytes in each packet can be used to carry the payload within the maximum transmission unit (MTU). In this context, network applications like remote procedure call (RPC) have to split their messages onto several packets, inevitably leading to severe end-to-end performance degradation. According to our experiments (see §II), the overhead of 48 bytes leads to a 25% increase in the flow completion time and a 20% decrease in end-to-end goodput. Such overhead is non-trivial.

Unfortunately, to the best of our knowledge, none of existing solutions is aware of the large per-packet byte overhead, which results in sub-optimal decisions. We classify existing solutions into two categories. (1) The *single-switch program deployment frameworks* [8, 4] optimize their program deployment on a *single* switch in order to minimize per-switch resource consumption. However, although these frameworks can be easily extended to support network-wide deployment, they inherently

2575-8411/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDCS54860.2022.00043

produce sub-optimal decisions. (2) The *network-wide program deployment frameworks* [9, 5, 6, 7] propose different strategies that seek for achieving various optimization objectives such as maximizing end-to-end performance. However, as mentioned above, they heavily rely on the packet header space to deliver metadata among switches, leading to non-trivial overheads.

To fill this gap, we propose Hermes, a novel network-wide program deployment framework that minimizes the per-packet byte overhead in inter-switch coordination. Specifically, for each input program, Hermes first enumerates every pair of the MATs defined in the program to obtain all the execution dependencies. According to these dependencies, it analyzes the amount (in bytes) of metadata required to be delivered between every pair of MATs. Next, it integrates its analysis results into its optimization objective of minimizing the maximum amount of the metadata carried by each packet between arbitrary two programmable switches. It also encodes switch resource limitations as constraints. With the objective and constraints, it formulates the problem of network-wide program deployment via mixed-integer linear programming (MILP). However, we prove that the MILP problem is NP-hard, making it unsolvable in polynomial time. As a result, the problem solving in large-scale networks remains time-consuming. In response, we also design a greedy-based heuristic in Hermes that sacrifices a small portion of optimality to retain timeliness. Thus, Hermes allows administrators to make tradeoffs between the per-packet byte overhead and timeliness based on the problem scale.

We have implemented a Hermes prototype on 32×100 Gbps Tofino programmable switches [10]. We build a real-world testbed and a large-scale simulator to evaluate Hermes. The experimental results show that compared to existing solutions, Hermes can reduce the per-packet byte overhead by 156 bytes. Also, the heuristic of Hermes makes near-optimal decisions while achieving orders-of-magnitude lower execution time.

II. Background and Motivation

A. Network-Wide Data Plane Program Deployment

Program deployment. We consider a common network scenario that corresponds to modern production networks such as data center networks [1, 2, 5]. This scenario has a control plane and a data plane. In the control plane, a cluster of controllers manages the data plane and offers the global network view to administrators. With this view, administrators determine their management intent (e.g., load balancing) consisting of various packet processing logic (e.g., packet routing). They implement their logic with the MATs offered by network programming languages like P4 [11] while specifying MAT dependencies in data plane programs to compose the complete intent. Next, they configure underlying programmable switches with their programs via *program deployment*, i.e., compiling programs to corresponding switch configurations and loading these configurations on switches. In the data plane, programmable switches process incoming packets with respect to the packet processing logic specified by switch configurations.

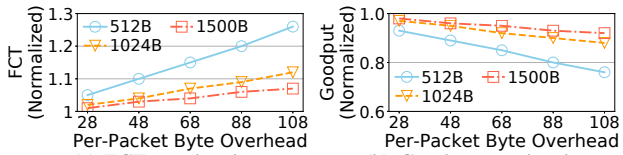
TABLE I: Common metadata in data plane programs.

Metadata	Size per switch	Common Usage in Data Plane Programs
Switch identifier	4 bytes	path tracing [17, 18, 19], path conformance [20, 21]
Queue lengths	6 bytes	congestion control [22, 1, 23, 24]
Timestamps	12 bytes	troubleshooting [17, 25, 21], anomaly detection [12, 26]
Counter index	4 bytes	hash tables [25, 27, 28, 29], sketches [30, 31, 32, 33]

Why network-wide deployment? The deployment requires the target switch to provide sufficient resources to realize user-specified logic. However, a single programmable switch has severe resource limitations, e.g., the scarce memory capacity (typically less than 15 MB [3, 4]) that implements the structures of MATs. These limitations forbid the common deployment situation where a number of concurrent programs jointly require excessive resources. For example, in software-defined measurement (SDM) [12, 13, 14, 15, 16], administrators aim to measure various traffic statistics (e.g., per-flow counts). They describe some measurement algorithms (e.g., sketches [12]), each of which measures a specific type of statistics, and deploy them at the same time. However, although each algorithm consumes limited resources, the overall resource consumption of all algorithms easily exhausts the resource capacity of a single switch, leading to deployment failures [6].

In response, recent frameworks [5, 6, 7] adopt the paradigm of *network-wide program deployment*, comprising three steps. (1) *Program preprocessing*. They merge input programs into a merged one and transform it into a table dependency graph (TDG) [8], in which nodes and edges are MATs and MAT dependencies, respectively. The TDG is an intuitive representation of the packet processing logic of input programs. (2) *MAT placement*. These frameworks build an MILP problem that places every TDG node (i.e., MAT) on one of the underlying programmable switches. They solve the MILP problem via ILP solvers or heuristics. (3) *Inter-switch coordination*. They reconstruct TDG edges (i.e., MAT dependencies). They enumerate every TDG edge and identify which two MATs the edge is connected to. If the two MATs, a and b, run on different switches, they set the upstream MAT a to piggyback metadata on the header space of each packet. Then they route the packet to the next switch, in which b extracts metadata from each packet. This preserves the original MAT dependencies. Table I summarizes widely-used metadata.

The network-wide program deployment has been exploited in several distributed networking systems. In addition to SDM, we describe another two examples here. (1) In in-band network telemetry (INT) [34], the task of traffic monitoring is split and deployed on different switches. At runtime, each switch adds an INT header to each packet. It adds its metadata, such as switch identifiers and queue lengths, to the INT header. When the packet leaves the network, the egress switch extracts the INT header from each packet and sends it to the control plane for network event detection. (2) In network function virtualization (NFV) [35, 36, 37], the entire chain of network functions (NFs) (e.g., key-value caches [35]) are offloaded to the data plane to achieve high packet processing performance. Due to resource limitations, each switch often only executes a specific NF. The NF needs to piggyback its processing results on packets to deliver information to its downstream NFs.



(a) FCT overhead. (b) Goodput overhead.

Fig. 2: *Impact of the per-packet byte overhead on end-to-end performance in terms of FCT and goodput.*

B. Overhead Analysis of Inter-Switch Coordination

Per-packet byte overhead. However, such inter-switch coordination dedicates a portion of the header space in each packet to piggyback metadata, leading to the *per-packet byte overhead*. For example, in a 5-hop end-to-end DCN transmission, the size of INT headers easily exceeds 48 bytes [38], which consumes more than 3.2% of a 1500-byte packet. As a result, administrators have to modify network applications to shorten the payload size of their packets so that their packets carrying additional metadata can fit the MTU. As a compromise, the end-to-end performance will be degraded since applications require more packets to transfer their messages.

Impact on end-to-end performance. We evaluate the impact of the per-packet byte overhead on end-to-end performance via testbed experiments. We build a simple testbed with a Tofino-based switch [10] and two end-hosts. We use two switch ports to connect end-hosts. We adopt the following settings. (1) We consider the DCN scenario. Since a flow typically traverses five switches within a DCN [39], we set the switch to repeat layer-3 routing five times for each packet. (2) In an end-host, we generate a flow with 10^6 packets, and direct the flow to another host. We fix the packet size to 512 bytes, 1024 bytes, and 1500 bytes, respectively. We set 512 bytes following the traffic characteristics in DCNs [40]. We select 1024 bytes and 1500 bytes (i.e., the RDMA MTU and the Ethernet MTU). Also, we adaptively tune the MTU so that the sum of the original packet size and the overhead will not exceed MTU. (3) To quantify the end-to-end performance, we measure two metrics, including the flow completion time (FCT) and the goodput of the flow. The FCT is calculated as the end-to-end latency of the flow. (4) We vary the size of metadata added to each packet from 28 bytes to 108 bytes.

Figure 2 presents the average normalized results after 100 runs when compared to the case when packets do not need to carry metadata. It indicates that the per-packet byte overhead results in significant degradation in end-to-end performance. For example, the overhead of 68 bytes can result in a 15% increase in FCT and a 16% decrease in goodput. Thus, our results highlight that it is essential to reduce the per-packet byte overhead to maintain end-to-end performance.

III. Hermes Overview

Design goals. In this paper, we propose Hermes, a novel program deployment framework. Different from existing frameworks that incur non-trivial byte overheads on packets (see §II), Hermes reduces such overheads in its program deployment. Specifically, our design of Hermes has three main goals.

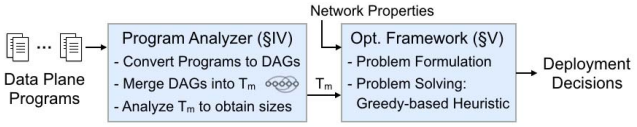


Fig. 3: *Overview of Hermes.*

- **Goal#1**: Given a set of data plane programs, Hermes aims to place every MAT invoked by these programs on a specific pipeline stage of underlying programmable switches.
- **Goal#2**: Hermes aims to retain the original MAT dependencies defined in the input programs in order to preserve the correctness of packet processing.
- **Goal#3**: Hermes also aims to minimize the per-packet byte overhead so as to maintain end-to-end performance.

Architecture and workflow. To achieve these goals, we propose the Hermes architecture comprising a program analyzer and an optimization framework. As shown in Figure 3, these components jointly perform a three-step workflow.

- **Step#1: Program analysis** (§IV). The program analyzer receives a set of data plane programs as input. It transforms each program into a TDG and merges all the TDGs into a merged one. Next, it enumerates every edge in the merged TDG. For the edge (a, b), in which the upstream MAT a passes metadata to the downstream MAT b, the analyzer calculates the size of metadata delivered by (a, b) and records this as a property of (a, b).
- **Step#2: Problem formulation** (§V-A-§V-C). The optimization framework takes both the compound TDG and network properties as input. It formulates the problem of program deployment as an MILP problem. In the MILP problem, the output is a set of decision variables indicating which switch stages each MAT defined in the compound TDG is placed on, which realizes the Goal#1. Next, the framework encodes switch resource limitations as the constraints of the MILP problem, thus guaranteeing the Goal#2 in its deployment. Also, it sets its optimization objective to minimizing the per-packet byte overhead, which satisfies the Goal#3.
- **Step#3: Problem solving** (§V-E). The optimization framework solves the MILP problem to produce its deployment decisions. In view of the NP hardness of the MILP problem (see §V-D), it leverages a Greedy-based heuristic that trades a small portion of optimality for timely problem solving instead of using the time-consuming ILP solvers. Our experiments in §VI demonstrate that the heuristic achieves the near-optimal results while reducing the execution time of problem solving by several orders of magnitude.

IV. Program Analyzer

Overview. In this section, we design a program analyzer in Hermes, which has two-fold goals. (1) The analyzer aims to transform the input set of data plane programs into a merged TDG T_m. The merged TDG T_m is a universal and intuitive representation of the packet processing logic defined in input programs and shields low-level program diversities (e.g., various ways of MAT invocation). With T_m, the optimization

TABLE II: Notation of symbols

Symbols for TDGs	
Φ	Set of input data plane programs.
T_m	Merged TDG $T_m = (V_{T_m}, E_{T_m})$.
V_{T_m}	Set of nodes in T_m (i.e., MATs).
E_{T_m}	Set of edges in T_m (i.e., MAT dependencies).
F_a^m	Set of matching fields used by the MAT a.
$\mathcal{A}_a$	Set of actions used by the MAT a.
F_a^a	Set of fields modified by the actions in $\mathcal{A}_a$.
$\mathcal{R}_a$	Set of user-specified rules in the MAT a.
$\mathcal{C}_a$	Maximum number of rules in $\mathcal{R}_a$.
$T(a,b)$	Type of the MAT dependency between a and b.
$A(a,b)$	Amount of metadata delivered from the MAT a to b.
$R(a)$	Resource requirement of the MAT a.
$R(a,i,u)$	Resource consumption of the MAT a on the i-th stage of u.
Symbols for network properties	
G	Network $G = (V_G, E_G)$.
V_G	Set of switches in the network G.
E_G	Set of links in the network G.
$\mathcal{P}(u,v)$	Set of paths between the switch u and the switch v.
Q	Number of switches. $Q = \lvert V_G \rvert$.
N	Number of links. $N = \lvert E_G \rvert$.
$P(u)$	Programmability of the switch u.
$\mathcal{C}_{stage}$	Number of stages in each switch.
$\mathcal{C}_{res}$	Per-stage resource capacity of each switch.
$t_s(u)$	Maximum transmission latency of the switch u.
$t_l(u,v)$	Transmission latency of the link (u,v).
$t_p(p)$	Transmission latency of the path p.
Symbols for auxiliary variables	
A_{max}	Max. amount of metadata delivered among two switches.
t_{e2e}	End-to-end per-packet transmission latency.
Q_{occ}	Number of occupied programmable switches.
ϵ_1, ϵ_2	Upper bound of objective values.
ρ	Number (i.e., identifier) of a specific switch stage.
$\mathcal{S}$	Set of programmable switches satisfying constraints.
Ω	Set of TDG segments.
$\mathcal{E}(a,p)$	Variable indicating if a switch or link a locates in the path p.
$\mathcal{L}(a,u)$	Variable indicating if the MAT a is deployed on the switch u.
Symbols for decision variables	
$x(a,i,u)$	Variable indicating if the MAT a is placed on the i-th stage of the switch u.
$y(u,v,p)$	Variable indicating if the switch u delivers its packets to the switch v via the path p.

framework of Hermes (§V) can focus on its optimization of per-packet byte overheads without the burden of handling irrelevant program-specific details. (2) The analyzer aims to profile the amount of metadata delivered between two arbitrary MATs in the merged TDG. Its analysis results are necessitated when encoding the optimization objective of program deployment (see §V-B). To achieve these goals, the analyzer executes two steps, including merging TDGs to T_m and analyzing T_m to obtain the properties of metadata sizes. Algorithm 1 summarizes the overall procedure of the analyzer.

Converting programs into TDGs (line 3). Given a set of input data plane programs, the program analyzer first leverages existing tools [41] to transform these programs into corresponding TDGs. Each TDG $T = (V_T, E_T)$ is a direct acyclic graph (DAG). Its nodes in the set V_T are the MATs defined in the original program while its directed edges in the set E_T correspond to MAT dependencies. Each MAT a in V_T has five properties, including (1) the set F_a^m of matching fields, which can be either header fields (e.g., source IP address) or metadata, (2) the set $\mathcal{A}_a$ of actions (e.g., modifying field values), (3) the set F_a^a of the header fields and metadata, which values are modified by the actions in $\mathcal{A}_a$, (4) the set $\mathcal{R}_a$ of user-specified rules indicating how to match packets (e.g., longest prefix match), which packets to match, and which actions in $\mathcal{A}_a$ to perform on the matched packets, and (5) its capacity $\mathcal{C}_a$, i.e., the maximum number of rules in $\mathcal{R}_a$.

Algorithm 1 Program Analyzer of Hermes

Input: Set Φ of input data plane programs
Output: The merged TDG $T_m = (V_{T_m}, E_{T_m})$
Variables: TDGs T_1, T_2, T_3, metadata field f

1: **function** TDG_MERGING(Φ)
2: Initialize $T_m = (V_{T_m}, E_{T_m})$ with empty sets
3: Convert the programs in Φ to TDGs
4: **while** Φ.size > 1 **do**
5: Extract two arbitrary TDGs, T_1 and T_2, from Φ
6: Merge T_1 and T_2 into T_3 via SPEED
7: Add T_3 to Φ
8: **return** the only TDG in Φ
9: **function** TDG_ANALYSIS(T_m)
10: **for** edge $(a,b) \in E_{T_m}$ **do**
11: **if** $T(a,b) = \mathbb{M}$ **then** ▷ Match dependency
12: $A(a,b) \leftarrow \sum_{\forall f \in F_a^a} \text{size}(f)$
13: **else if** $T(a,b) = \mathbb{A}$ **then** ▷ Action dependency
14: $A(a,b) \leftarrow \sum_{\forall f \in F_a^a \cup F_b^a} \text{size}(f)$
15: **else if** $T(a,b) = \mathbb{R}$ **then** ▷ Reverse match dependency
16: Continue
17: **else if** $T(a,b) = \mathbb{S}$ **then** ▷ Successor dependency
18: $A(a,b) \leftarrow \sum_{\forall f \in F_a^a} \text{size}(f)$
19: **return** T_m
20: **function** PROGRAM_ANALYZER(Φ)
21: $T_m \leftarrow$ TDG_MERGING(Φ)
22: $T_m \leftarrow$ TDG_ANALYSIS(T_m)
23: **return** T_m

Moreover, each directed edge in E_T indicates that two specific MATs in V_T are interdependent. More precisely, for the edge $(a,b) \in E_T$, a and b are two MATs in V_T while a is an upstream MAT of b. The edge has a type $T(a,b)$ [8], which can be: (1) match dependency ($\mathbb{M}$): b matches a field f, which value has been modified by a, i.e., $f \in F_a^a \cap F_b^m$, (2) action dependency ($\mathbb{A}$): a field f is modified by both a and b, i.e., $f \in F_a^a \cap F_b^a$; (3) reverse match dependency ($\mathbb{R}$): b modifies a field f matched by a, i.e., $f \in F_a^m \cap F_b^a$; and (4) successor dependency ($\mathbb{S}$): whether to invoke b depends on the values of the fields or metadata modified by a. In this context, if two MATs $a, b \in V_T$ are connected by an edge $(a,b) \in E_T$, the execution of b depends on the processing results of a so that b must be invoked after a during packet processing. Otherwise, the original packet processing logic will be violated.

TDG merging (lines 4-8). After transforming programs into TDGs, the analyzer starts to merge these TDGs into a merged TDG. In particular, as reported by previous studies [6, 42, 43], different programs exhibit *redundancy* in their packet processing logic. For example, in SDM, various measurement algorithms need to invoke the same functionality of calculating indexes via hash functions [12]. In this case, when merging these algorithms, we have the opportunity of reducing switch resource consumption by eliminating redundancy among these programs. Given this benefit, we borrow the same idea of SPEED [6] that implements the functionality of TDG merging in the analyzer. Specifically, given a set Φ of TDGs, the

analyzer randomly selects two arbitrary TDGs, T_1 and T_2, from Φ. It merges T_1 and T_2 to a merged TDG T_3 via SPEED, which executes three steps. (1) It identifies the redundant MATs, i.e., the MATs with the same properties, among T_1 and T_2. (2) It initializes T_3 by configuring the set V_{T_3} of MATs and the set E_{T_3} of MAT dependencies as $V_{T_1} \cup V_{T_2}$ and $E_{T_1} \cup E_{T_2}$, respectively. (3) It removes as many redundant MATs in V_{T_3} while preserving the edges in E_{T_3}. Next, it populates the merged TDG T_3 back to the set Φ. When Φ only contains one TDG, the analyzer returns the TDG as T_m.

TDG analysis (lines 9-19). The analyzer performs TDG analysis on the merged TDG T_m to acquire the amount of metadata delivered between an arbitrary pair of interdependent MATs. To achieve this goal, we design the analyzer to enumerate every edge of the merged TDG T_m in E_{T_m}. For the edge $(a, b) \in E_{T_m}$, it locates the two MATs, i.e., the upstream MAT a and the downstream MAT b, the edge connects with in V_{T_m}. Then it calculates the amount $A(a, b)$ of metadata delivered by (a, b) with respect to the edge type $T(a, b)$. (1) If $T(a, b) = \mathbb{M}$, a passes its processing results in F_a^a to b, which matches some fields in F_a^a. In this case, the analyzer identifies which fields in F_a^a are metadata fields and accumulates their sizes as $A(a, b)$. Note that other fields are header fields, which already reside in packets and have no impact on per-packet byte overheads so that we do not take them into account. (2) If $T(a, b) = \mathbb{A}$, a and b modify the same fields in $F_a^a \cup F_b^a$. Thus, the analyzer calculates $A(a, b)$ as the sum of the size of each metadata field in $F_a^a \cup F_b^a$. (3) If $T(a, b) = \mathbb{R}$, b does not depend on the processing results of a so that the analyzer ignores this case. (4) If $T(a, b) = \mathbb{S}$, the processing results of a determine whether to execute b so that the analyzer returns the sum of the size of metadata fields in F_a^a as $A(a, b)$. Next, the analyzer records $A(a, b)$ in T_m as a property of (a, b).

V. Optimization Framework

Overview. In this section, we present the design details of the optimization framework in Hermes that deploys the merged TDG T_m onto the substrate network comprising traditional switches and programmable switches. Specifically, the framework takes both T_m and the properties of network devices as input. It aims to produce a set of decision variables, each of which specifies whether an MAT in T_m is placed on a specific switch or not (§V-A). It also aims to minimize the per-packet byte overhead in the inter-switch communication (§V-B). Meanwhile, it encodes switch resource limitations as constraints to ensure that its deployment faithfully preserves those limitations (§V-C). However, we prove that the above problem is NP-hard, making its solving challenging (§V-D). Thus, the framework offers a greedy-based heuristic that seeks for near-optimal deployment in a timely manner (§V-E).

A. Input and Output

Input: merged TDG and network properties. The optimization framework takes both the merged TDG T_m, which is the universal representation of input data plane programs (see §IV), and network properties as input. The merged TDG T_m comprises the set V_{T_m} of the nodes of T_m (i.e., MATs) and the set E_{T_m} of the edges of T_m (i.e., MAT dependencies). The properties of nodes and edges have been specified in §IV. Next, we consider a common network that can be represented by a indirected graph $G = (V_G, E_G)$, where V_G denotes the sets of switches while E_G includes the links between switches. The network comprises $Q = |V_G|$ switches and $K = |E_G|$ links. Each switch $u \in V_G$ has four properties: (1) $P(u)$ represents the programmability of u. If u is programmable, $P(u) = 1$; otherwise, $P(u) = 0$. (2) Each programmable switch u ($P(u) = 1$) has a finite number $\mathcal{C}_{stage}$ of packet processing stages. (3) Each stage of a programmable switch u ($P(u) = 1$) has a finite capacity of resources. Note that a stage has various types of resources such as SRAM, TCAM, and arithmetic logic units (ALUs). Without losing generality, we use a single variable $\mathcal{C}_{res}$ to represent the overall capacity of per-stage resources for simplicity. (4) Each switch u has a maximum transmission latency $t_s(u)$ (in microseconds). Moreover, for each link $l = (u, v) \in E_G$ ($u, v \in V_G$), we consider its transmission latency $t_l(u, v)$ (in microseconds). Note that the properties of latency statistics can be easily obtained by means of a central controller [44, 45].

According to its input, the framework further creates a set $\mathcal{P}(u, v)$ that contains all the paths between two arbitrary switches, u and v ($u, v \in V_G$). Each path is a sequence of links and switches. Here, we use a variable $\mathcal{E}(a, p)$ to specify if a link or a switch $a \in E_G \cup V_G$ locates in a path $p \in \mathcal{P}(u, v)$. If so, $\mathcal{E}(a, p) = 1$; otherwise, $\mathcal{E}(a, p) = 0$. Each path $p \in \mathcal{P}(u, v)$ exhibits the transmission latency $t_p(p)$ (in microseconds), which is the sum of transmission latency of all the links and switches in p, i.e.

$$t_p(p) = \sum_{\forall u \in V_G} \mathcal{E}(u, p) \cdot t_s(u) + \sum_{\forall l \in E_G} \mathcal{E}(l, p) \cdot t_l(l)$$

Output: program deployment decisions. The optimization framework produces two sets of decision variables. First, each variable $x(a, i, u)$ in the set $\{x(a, i, u)\}$ specifies whether the MAT $a \in V_{T_m}$ is deployed on the i-th stage of the switch $u \in V_G$. If so, $x(a, i, u) = 1$; otherwise, $x(a, i, u) = 0$. Second, each variable $y(u, v, p)$ in the set $\{y(u, v, p)\}$ determines whether the switch u delivers its packets to the switch v via the path p. If so, $y(u, v, p) = 1$; otherwise, $y(u, v, p) = 0$.

B. Optimization Objective

The optimization framework has three objectives, including (1) minimizing the per-packet byte overhead, (2) minimizing the per-packet transmission latency, and (3) minimizing the number of occupied programmable switches. The first two objectives are used to ensure the end-to-end performance while achieving the third objective enhances resource efficiency. The framework simultaneously integrates these objectives into its problem formulation. We elaborate the details as follows.

Obj#1: Minimizing per-packet byte overhead. The main objective of Hermes is to minimize the per-packet byte overhead. More precisely, the framework aims to minimize the maximum

amount A_{max} of metadata required to be delivered between an arbitrary pair of programmable switches, i.e.

$$\min\ A_{max},$$
$$A_{max} = \max(\sum_{\forall a,b \in V_{T_m}, a \neq b} \sum_{\forall u,v \in V_G} \sum_{\forall i \in [1,\mathcal{C}_{stage}]} \sum_{\forall j \in [1,\mathcal{C}_{stage}]} P(u) \cdot P(v) \cdot x(a,i,u) \cdot x(b,j,v) \cdot A(a,b)) \quad (1)$$

Obj#2: Minimizing per-packet transmission latency. Also, we consider the second objective of minimizing the per-packet transmission latency. The latency equals the sum t_{e2e} of the latency of the paths each packet traverses, i.e.

$$\min\ t_{e2e},$$
$$t_{e2e} = \frac{1}{2} \cdot \sum_{\forall u,v \in V_G, u \neq v} \sum_{\forall i \in [1,\mathcal{C}_{stage}]} \sum_{\forall j \in [1,\mathcal{C}_{stage}]} \sum_{\forall p \in \mathcal{P}(u,v)} P(u) \cdot P(v) \cdot x(a,i,u) \cdot x(b,j,v) \cdot y(u,v,p) \cdot t_p(p) \quad (2)$$

Obj#3: Minimizing number of occupied switches. The third objective is to minimize the number Q_{occ} of programmable switches occupied by the program deployment, i.e.

$$\min\ Q_{occ},$$
$$Q_{occ} = \sum_{\forall u \in V_G} \sum_{\forall i \in [1,\mathcal{C}_{stage}]} P(u) \cdot x(a,i,u) \quad (3)$$

Integration of multiple objectives. The framework adopts the ϵ-constraint method [46] that transforms the multi-objective problem into a single-objective one so as to simultaneously optimize multiple objectives within the same problem. More precisely, it selects the objective of minimizing the per-packet byte overhead while transforming other objectives into two constraints in the single-objective problem. Thus, the framework obtains the single-objective MILP problem **P#1** as:

$$\textbf{P\#1}: \min\ A_{max}$$
$$\text{s.t. } t_{e2e} \leq \epsilon_1, \quad (4)$$
$$Q_{occ} \leq \epsilon_2, \quad (5)$$
$$\text{Other constraints in §V-C}$$

where ϵ_1 and ϵ_2 are the upper bounds of t_{e2e} and Q_{occ}, respectively. The framework allows administrators to flexibly submit their desired bounds on demand. For example, in view of strict service-level agreements (SLAs), administrators can set ϵ_1 to a small value of a few microseconds [47, 48].

C. Constraints

The optimization framework further encodes three additional types of constraints in its problem formulation.

Node deployment. Each MAT in V_{T_m} should be deployed on at least one stage of programmable switches in the substrate network, i.e.

$$\sum_{\forall u \in V_G} \sum_{\forall i \in [1,\mathcal{C}_{stage}]} x(a,i,u) \geq 1, \forall a \in V_{T_m} \quad (6)$$

Edge deployment. Each MAT dependency in E_{T_m} should be faithfully maintained after MAT deployment. More precisely, given an arbitrary dependency $(a,b) \in E_{T_m}$, the framework encodes two constraints.

(1) When the two MATs, a and b, are deployed on different switches, the switch u that runs a should be the upstream of the switch v that runs b. In other words, u should deliver its packets to v through at least one path at runtime, i.e.

$$\sum_{\forall p \in \mathcal{P}(u,v)} y(u,v,p) \geq 1, \forall u,v \in V_G, \forall a,b \in V_{T_m},$$
$$\mathcal{L}(a,u) \cdot \mathcal{L}(b,v) = 1 \quad (7)$$

where the boolean variable $\mathcal{L}(a,u)$ indicates if the MAT a is deployed on the switch u.

$$\mathcal{L}(a,u) = \begin{cases} 1, & \sum_{\forall i \in [1,\mathcal{C}_{stage}]} x(a,i,u) \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

(2) When the two MATs are placed on the same switch, a should be invoked before b. In other words, for each incoming packet, the final stage that deploys (a portion of) a should be invoked before the first stage that deploys (a portion of) b, i.e.

$$\rho_{end}(a) < \rho_{begin}(b), \forall (a,b) \in E_{T_m},$$
$$x(a,i,u) \cdot x(b,j,u) = 1, \forall i,j \in [1,\mathcal{C}_{stage}] \quad (8)$$

where $\rho_{end}(a)$ refers to the number of the last stage that runs (a portion of) a while $\rho_{begin}(b)$ indicates the number of the first stage that deploys (a portion of) b.

Switch resource limitations. The total amount of resources consumed by all the MATs placed on a specific switch stage should always be less than the per-stage resource capacity, i.e.

$$\sum_{\forall a \in V_{T_m}} x(a,i,u) \cdot R(a,i,u) \leq \mathcal{C}_{res},$$
$$\forall u \in V_G, \forall i \in [1,\mathcal{C}_{stage}] \quad (9)$$

where $R(a,i,u)$ denotes the resource consumption of the MAT a in the i-th stage of the switch u. Note that the sum $R(a)$ of $R(a,i,u)$ in all the stages of the switch u, i.e., $R(a) = \sum_{\forall i \in [1,\mathcal{C}_{stage}]} R(a,i,u)$, equals the resource requirement of the MAT a, which can be calculated via the static code analysis on MAT properties (e.g., $\mathcal{C}_a$) [8, 49].

D. Challenge of Problem Solving

However, we prove that the above MILP problem is NP-hard, making problem solving challenging.

Theorem 1. *The problem **P#1** is NP-hard.*

Proof. We consider a special case of **P#1**. This case simplifies our problem via three restrictions. First, it restricts that the amount of metadata delivered between an arbitrary pair of MATs always equals one byte, i.e., $A(a,b) = 1$ ($\forall a,b \in V_{T_m}$). Thus, it transforms the objective into minimizing the total number of stages occupied by MATs. Second, it restricts that the upper bounds of t_{e2e} and Q_{occ}, i.e., ϵ_1 and ϵ_2, are unlimited, i.e., $\epsilon_1 = \epsilon_2 = +\infty$. Third, it restricts that the network only has one programmable switch, which relaxes the Equation (7). These restrictions reduce **P#1** to the NP-hard bin packing problem [50]. Thus, Theorem 1 follows. □

Algorithm 2 Greedy-based Heuristic of Hermes

Input: $T_m = (V_{T_m}, E_{T_m})$, $G = (V_G, E_G)$, ϵ_1, ϵ_2
Output: Decision variables in $\{x(a,i,u)\}$ and $\{y(u,v,p)\}$
Variables: Set $\mathcal{S}$ of switches, set Ω of TDG segments, min amount A_{min}

1: **function** SPLIT_TDG(T_m, Ω)
2: **if** $\sum_{\forall a \in V_{T_m}} R(a) \leq \mathcal{C}_{stage} \cdot \mathcal{C}_{res}$ **then**
3: **return** $\Omega \cup \{T_m\}$ ▷ T_m satisfies switch resource limitations
4: Sort V_{T_m} via topological sorting
5: Initialize $T_m^a = (V_{T_m^a}, E_{T_m^a})$ and $T_m^b = (V_{T_m^b}, E_{T_m^b})$
6: Initialize $V_a, V_b \leftarrow \emptyset, V_{T_m}$
7: Initialize $A_{min} \leftarrow +\infty$
8: **for** $a \in V_b$ **do** ▷ Search the splitting location in T_m
9: Move a from V_b to V_a
10: **if** $\sum_{\forall a \in V_a} \sum_{\forall b \in V_b} A(a,b) < A_{min}$ **then**
11: $A_{min} \leftarrow \sum_{\forall a \in V_a} \sum_{\forall b \in V_b} A(a,b)$
12: $V_{T_m^a}, V_{T_m^b} \leftarrow V_a, V_b$
13: $E_{T_m^a} \leftarrow \{(a,b) | (a,b) \in E_{T_m}, a,b \in V_a\}$
14: $E_{T_m^b} \leftarrow \{(a,b) | (a,b) \in E_{T_m}, a,b \in V_b\}$
15: $\Omega_a \leftarrow$ SPLIT_TDG(T_m^a, Ω)
16: $\Omega_b \leftarrow$ SPLIT_TDG(T_m^b, Ω)
17: **return** $\Omega_a \cup \Omega_b$
18: **function** GREEDY_BASED_HEURISTIC(T_m, G, ϵ_1, ϵ_2)
19: Initialize $\{x(a,i,u)\}$ and $\{y(u,v,p)\}$ with zero
20: $\Omega \leftarrow$ SPLIT_TDG(T_m, $\emptyset$)
21: **for** $u \in V_G$ **do**
22: **if** $P(u) == 1$ **then** ▷ u is programmable
23: $\mathcal{S} \leftarrow$ SELECT_SWITCHES(u, G, ϵ_1, ϵ_2)
24: **if** $|\Omega| \leq |\mathcal{S}|$ **then** ▷ There are enough switches to place T_m
25: Set $\{x(a,i,u)\}$ to map segments in Ω to switches in $\mathcal{S}$
26: **for** $i \in [1, |\mathcal{S}| - 1]$ **do**
27: Locate the i-th switch u and the $(i+1)$-th switch v
28: Locate the shortest path p between u and v
29: $y(u,v,p) \leftarrow 1$
30: **return** $\{x(a,i,u)\}$ and $\{y(u,v,p)\}$

Although this problem can be solved through commodity ILP solvers such as Gurobi [51], the scalability of ILP solvers is limited. When the network scale (i.e., the number of switches and links) grows, the execution time of ILP solvers becomes non-trivial. This is because ILP solvers need to enumerate an exponential number of variables and constraints. More precisely, given a substrate network $G = (V_G, E_G)$, the total number of decision variables in $\{x(a,i,u)\}$ and $\{y(u,v,p)\}$ is at least $2^{|V_G|} + 2^{|V_G| \cdot |E_G|}$ while the number of constraints is $O((|V_G|)^2 \cdot |E_G|)$. In the worst case, ILP solvers enumerate every decision variable while explicitly checking all the constraints for each of their enumerations. This yields the time complexity of $O((2^{|V_G|} + 2^{|V_G| \cdot |E_G|}) \cdot (|V_G|)^2 \cdot |E_G|)$. Given this exponential complexity, it is impractical for ILP solvers to converge within polynomial time.

E. Problem Solving: Greedy-based Heuristic

Overview. We design a greedy-based heuristic in the framework for timely and near-optimal problem solving. The key idea of the heuristic is intuitive but effective: when deploying TDG edges (i.e., MAT dependencies), the heuristic prefers to deploy the MATs corresponding to the edges, which deliver larger amounts of metadata (i.e., $A(a,b)$) than other edges, on the same programmable switch. Thus, only the interdependent MATs corresponding to the edges with small values of $A(a,b)$ will be deployed in the inter-switch manner. In this context, the amount of metadata carried by each packet in the inter-switch coordination remains small, thus reducing the per-packet byte overhead and ensuring end-to-end performance.

Workflow. Algorithm 2 shows the workflow of the greedy-based heuristic. Specifically, it first initializes the decision variables in $\{x(a,i,u)\}$ and $\{y(u,v,p)\}$ with zero (line 10). Next, it starts to split the TDG T_m into a set Ω of several TDG segments (line 11). Each TDG segment satisfies switch resource limitations and can be fully deployed on a single switch (lines 2-3). More precisely, the heuristic adopts the divide-and-conquer paradigm that decomposes the entire T_m into two segments and splits the two segments in turn until their segments can satisfy switch resource limitations (lines 15-16).

In particular, when splitting T_m into two segments, T_m^a and T_m^b, the heuristic uses a greedy-based strategy. The strategy first initializes two sets, V_a and V_b, of MATs with the empty set and V_{T_m}, respectively (lines 6). It also records the minimum amount of metadata delivered by the MATs in V_a to that in V_b as A_{min}. Then it enumerates every MAT in V_b. It moves the current MAT a from V_b to V_a and determines whether the total amount of metadata delivered between the two sets of MATs is less than A_{min} (lines 9-10). If so, it updates A_{min} and records V_a and V_b as the sets of MATs of the segments T_m^a and T_m^b (lines 11-12). In this context, after enumeration, the heuristic obtains T_m^a and T_m^b with the minimum amount of metadata delivered between the MATs in the two segments.

After splitting T_m, the heuristic enumerates every switch. For the switch $u \in V_G$, it determines if u is programmable or not (line 22). If not, it turns to the next switch. Otherwise, it searches the $\epsilon_2 - 1$ closet programmable switches, each of which has at least one path connecting to u, while the latency of passing through these switches is less than ϵ_1 (line 23). It treats these switches as the candidates of deploying T_m because selecting them satisfies the constraints of end-to-end transmission latency and the number of occupied switches (see Equations (4)-(5)). It records those switches in a set $\mathcal{S}$. Then it determines whether there are enough switches in $\mathcal{S}$ to deploy the TDG segments in Ω (line 24). If not, it starts its next enumeration. Otherwise, it enumerates every segment in Ω and deploys it on a specific candidate switch in $\mathcal{S}$ (line 25). Also, the heuristic setups the inter-switch coordination by ordering two adjacent programmable switches that run TDG segments to communicate via the shortest path between them (i.e., the path with the minimum transmission latency) (lines 26-29).

Theorem 2. *Given $T_m = (V_{T_m}, E_{T_m})$ and a network $G = (V_G, E_G)$, the worst-case time complexity of Algorithm 2 is $O((|V_{T_m}| + |E_{T_m}|) \cdot log\,|V_{T_m}| + |V_G|^2)$.*

Proof. Algorithm 2 splits T_m (lines 10-11), which worst-case time complexity is $O((|V_{T_m}| + |E_{T_m}|) \cdot log\,|V_{T_m}|)$. Then it enumerates every switch in V_G. In each enumeration, it places TDG segments on programmable switches and reconstructs MAT dependencies (lines 15-20), leading to the complexity of $O(|V_G|)$. Also, its other operations can be completed within $O(|V_G|)$ time. Thus, the time complexity of Algorithm 2 is $O((|V_{T_m}| + |E_{T_m}|) \cdot log\,|V_{T_m}| + |V_G| \cdot (|V_G| + |V_G|)) = O((|V_{T_m}| + |E_{T_m}|) \cdot log\,|V_{T_m}| + |V_G|^2)$. □

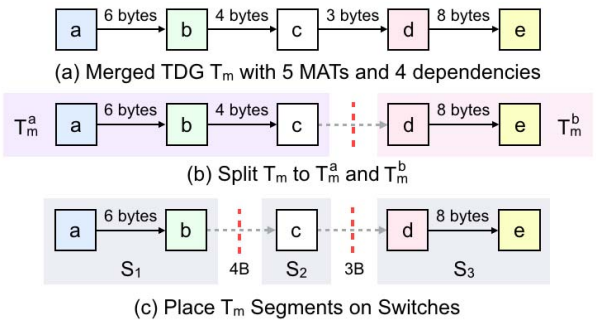


Fig. 4: *Example of the greedy-based heuristic in Hermes.*

Example. Figure 4 presents an example of deploying a merged TDG T_m, which properties are shown in Figure 4(a), via the greedy-based heuristic. In this example, we assume that the heuristic has already populated three candidate switches into $\mathcal{S} = \{S_1, S_2, S_3\}$ while each switch can only tolerate two MATs due to its resource limitations. In this context, the heuristic starts to split T_m into several segments. It first determines whether the entire T_m can be deployed onto one single switch. Since T_m has five MATs, which exceeds the per-switch capacity of two MATs, the heuristic splits T_m into both T_m^a with $V_{T_m^a} = \{a, b, c\}$ and T_m^b with $V_{T_m^b} = \{d, e\}$. The amount of metadata delivered from the MATs in $V_{T_m^a}$ to that in $V_{T_m^b}$ is 3 bytes, which is minimum. As shown in Figure 4(b), T_m^a contains three MATs, which still cannot be fitted into a switch. To this end, the heuristic starts the next split that splits T_m^a into two segments, in which MATs transfer the minimal amount of 4 bytes metadata. After that, it obtains three TDG segments that are compatible with switch resource limitations (Figure 4(c)). It deploys the three segments on the three switches in $\mathcal{S}$, respectively, while setting the inter-switch coordination. As a result, the maximum per-packet byte overhead equals 4 bytes. Such overhead is smaller than the overheads incurred by other deployment solutions, e.g., 8 bytes when placing a and b on S_1, c and d on S_2, and e on S_3.

VI. Evaluation

In this section, we evaluate Hermes at scale. We present the average after 100 runs. We showcase our findings as follows.

- (Exp#1) When deploying ten real programs on our testbed, Hermes reduces the per-packet overhead by up to 156 bytes.
- (Exp#2) In our large-scale simulation, Hermes decreases the per-packet byte overhead by up to 34% compared to other solutions when deploying 50 concurrent programs.
- (Exp#3) When simultaneously deploying 50 programs on the network comprising hundreds of switches and links, Hermes achieves several orders-of-magnitude lower execution time than ILP-based frameworks.
- (Exp#4) Compared to other solutions, Hermes reduces the overheads on end-to-end performance by up to 145%.
- (Exp#5) Even when the number of concurrent programs to be deployed increases, Hermes retains high scalability in terms of lower overheads and execution time and higher end-to-end performance compared to other solutions.
- (Exp#6) Hermes does not use additional switch resources.

TABLE III: Topologies used by our experiments.

Topology ID	1	2	3	4	5	6	7	8	9	10
# of Nodes	62	70	66	75	73	65	68	69	74	69
# of Edges	62	85	78	74	70	79	92	74	92	98

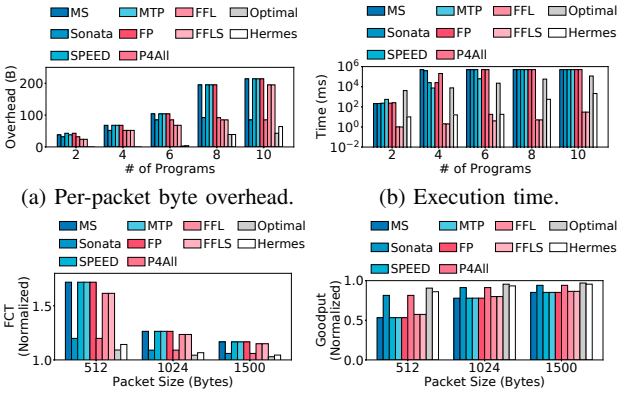
A. Experimental Setup

Implementation. Our implementation consists of a program analyzer and an optimization framework. For the program analyzer, we implement its functionalities of converting programs into TDGs and merging TDGs on P4C [41] and SPEED [6], respectively, while realizing its TDG analysis in C++. Also, we implement the framework in C++. Next, we implement a backend of Hermes that takes the decision variables of Hermes as input and transforms these variables into corresponding switch configurations. More precisely, according to the variables, the backend determines which MATs and MAT dependencies to be deployed on a target switch. It inputs them into an off-the-shelf switch compiler offered by switch vendors, which compiles its input to a binary that can be loaded to commodity programmable switches. At runtime, it invokes the network controller to direct traffic to correctly pass through a sequence of programmable switches based on the decision variables, which maintains the inter-switch coordination.

Testbed and simulator. We build a testbed comprising three 32×100 Gbps Tofino programmable switches [10]. We connect the three switches via 100 Gbps links to compose a linear topology. We connect two servers to the two switches at the edges of our testbed, respectively. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU (2.60 GHz) and 128 GB RAM. We run PktGen [52] with DPDK 19.05 [53] on the two servers. We activate PktGen in one server to generate traffic while using another one as the receiver. Moreover, as we are unable to construct large-scale networks, we build a simulator in C++ to simulate these networks and evaluate Hermes at scale. The simulator runs on a server, which configurations are the same as mentioned above. It takes the graph of a network topology as input and generates the corresponding simulated network for Hermes to make decisions. It allows us to flexibly tune network properties such as the number of programmable switches during our experiments. We illustrate this as follows.

Network topologies. In our simulation (Exp#4-6), we select 10 real-world wide-area network (WAN) topologies from [54]. Table III shows the scale of these topologies. We adopt practical settings [55, 56, 6, 57] to configure their properties. More precisely, we randomly choose 50% switches as programmable switches, each of which is configured with the same settings as Tofino switches. The transmission latency $t_s(u)$ of each switch is set to 1 μs while the latency $t_l(u, v)$ of a link is set to be randomly distributed between 1 ms to 10 ms.

Data plane programs. In our experiments, we use ten real programs. Each program corresponds to a specific version of switch.p4 [58], which settings are illustrated in [6]. Moreover, we generate additional 40 synthetic programs with respect to the features of real-world programs. In each synthetic program, we set the normalized per-stage resource consumption of each



(a) Per-packet byte overhead. (b) Execution time.

(c) Impact on FCT. (d) Impact on goodput.

Fig. 5: (Exp#1) *Testbed experiments.*

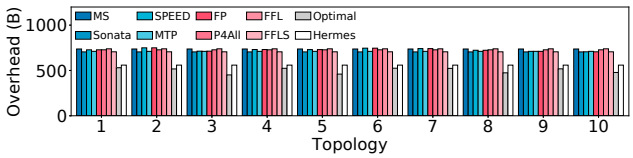


Fig. 6: (Exp#2) *Per-packet byte overhead in the simulation.*

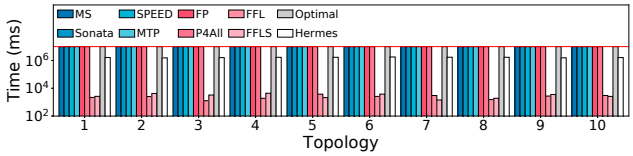


Fig. 7: (Exp#3) *Execution time in the simulation. For the cases where the execution time exceeds two hours, we set the bar height to* 10^7 *ms due to limited figure sizes.*

MAT to be uniformly distributed between 10% and 50%. We set the number of MATs in each program to be randomly distributed between 10 and 20. We set each pair of MATs to have a dependency with a probability of 30%.

Comparison solutions. We compare Hermes with two classes of frameworks. We implement the first class with ILP solvers. This class contains Min-Stage (MS) [8] and Sonata [4] that deploy each program on a single switch, SPEED [6] that enables network-wide program deployment while optimizing packet processing performance, MTP [57] that enhances SPEED with the capability of avoiding control plane overload, Flightplan (FP) [7] that supports program deployment on heterogeneous devices, and P4All [59] that allows modular programming while shielding programmers from low-level deployment details. Moreover, the second class comprises two heuristic-based solutions, i.e., first fit by level (FFL) and first fit by level and size (FFLS) [8, 6]. The two solutions adopt the greedy strategies of FFL and FFLS to deploy a program on a single switch, respectively. In particular, for a fair comparison, we implement the first class of frameworks using the same ILP solver, Gurobi [51]. Also, we extend Min-Stage, Sonata, FFL, and FFLS, to deploy input programs on switches one by one. We also implement an Gurobi-based version of Hermes ("Optimal"). Thus, we can evaluate the greedy-based heuristic by comparing its results with the optimal results.

Other settings. Due to the space limitation, our experiments focus on the per-packet byte overhead and the impact on the end-to-end performance. Therefore, we relax the constraints of transmission latency and the number of occupied switches with loose ϵ_1 and ϵ_2. However, our results indicate that Hermes faithfully preserves these constraints in all cases. We plan to present more results in the near future.

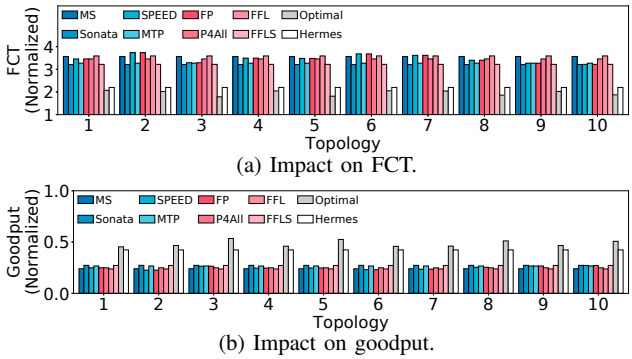
B. Experimental Results

(Exp#1) Testbed experiments. We measure the per-packet byte overhead in our testbed to demonstrate the effectiveness of Hermes. We use Hermes and the comparison solutions to simultaneously deploy multiple data plane programs on testbed switches, respectively. We vary the number of concurrent programs from 2 to 10. Then we measure the maximum amount of metadata delivered between an arbitrary pair of testbed switches as the per-packet byte overhead. Other settings are the same as §II-B. As shown in Figure 5(a), Hermes reduces the overhead by up to 156 bytes when compared with other solutions. The reason is that Hermes encodes its objective as minimizing the per-packet byte overhead, while other solutions inherently fail to optimize such overheads. The results also indicate that compared to Optimal, the greedy-based heuristic of Hermes achieves the optimal results as the scale of our testbed is small. Thus, as shown in Figure 5(c)-(d), Hermes avoids significant overheads in the FCT and goodput. Also, its execution time remains acceptable. For example, in the worst case, it completes within 2.1 s, which is orders-of-magnitude lower than the execution time of ILP-based solutions.

(Exp#2) Per-packet byte overhead in the simulation. We evaluate Hermes at scale with our simulator. We use Hermes and the comparison solutions to simultaneously deploy 50 programs on each simulated network, respectively. We adopt the same settings as Exp#1. Figure 6 indicates that compared to other solutions, Hermes reduces the per-packet byte overhead by up to 34% Also, the results show that the greedy-based heuristic of Hermes obtains the near-optimal results. For example, in the 10-th network, the overhead of the Hermes heuristic is only 16% higher than the optimal results.

(Exp#3) Execution time in the simulation. We measure the execution time of Hermes in Exp#2. Figure 7 presents the results indicating that the Hermes heuristic achieves orders-of-magnitude lower execution time than the ILP-based frameworks. The reason is that the ILP-based frameworks invoke ILP solvers for problem solving, which inevitably suffers from the exponential time of enumerating every possible condition. Compared to them, although Hermes sacrifices a small portion of optimality, it reduces the search space of problem solving via its greedy-based strategy so that it achieves polynomial time complexity. Thus, its execution time remains acceptable for offline deployment. Note that although FFL and FFLS converge faster, they ignore the per-packet byte overhead so that their deployment is evitably sub-optimal.

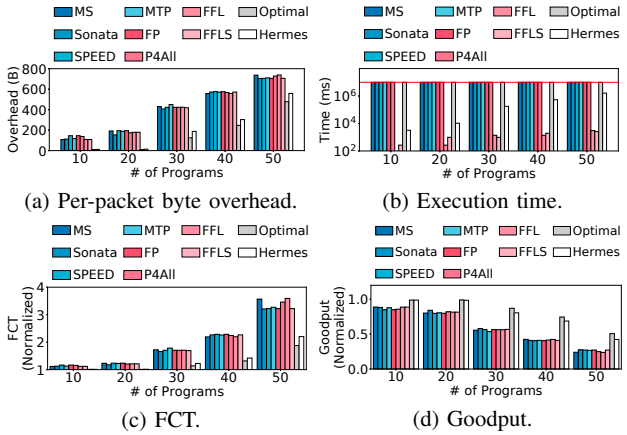
(Exp#4) Impact on the end-to-end performance. We also evaluate the impact of Hermes on the end-to-end performance



(a) Impact on FCT.

(b) Impact on goodput.

Fig. 8: (Exp#4) *Impact on the end-to-end performance.*



(a) Per-packet byte overhead.

(b) Execution time.

(c) FCT.

(d) Goodput.

Fig. 9: (Exp#5) *Scalability. For the cases where the execution time exceeds two hours, we set the bar height to* 10^7 *ms due to limited figure sizes.*

in terms of FCT and goodput. Recall from §II-B that higher per-packet byte overheads lead to a non-trivial performance reduction. Since Hermes reduces the per-packet byte overhead through its program deployment, it avoids significant performance overheads. We validate our claim by measuring the performance of 1024-byte packets during Exp#2. Figure 8 indicates that Hermes reduces the overheads of other solutions by up to 145%, which validates our claim.

(Exp#5) Scalability. We quantify the scalability of Hermes. To do this, we vary the number of programs to be simultaneously deployed on the 10-th topology from 10 to 50 in our simulator. Figure 9 presents the impact of program number on the per-packet byte overhead, the execution time, and the FCT and goodput. It indicates that compared to other frameworks, the execution time of Hermes gradually increases as the program number increases while Hermes retains lower overheads in all cases. Thus, the results validate the high scalability of Hermes.

(Exp#6) Switch resource consumption. One concern may be that Hermes may require excessive switch resources. Thus, we quantify the impact of Hermes on switch resource consumption. More precisely, we consider the scenario of software-defined measurement [12, 14, 15]. We randomly choose ten sketches and use SPEED and Hermes to deploy these sketches on testbed switches at the same time, respectively. Also, we deploy each sketch on a single switch and measure its resource consumption. Then we accumulate the resource consumption of the ten sketches, i.e., the consumption when the inter-switch coordination is not activated, as the ground true. By comparing the consumption of the sketches deployed by Hermes with the ground true, we obtain the resource consumption of the inter-switch coordination. Our results indicate that in addition to the original resource consumption of realizing the inter-switch coordination, Hermes does not use additional switch resources. This is because our design does not insert any additional logic to switches, thus maintaining resource efficiency.

VII. Related Work

Overhead optimization. Researchers have proposed a number of solutions that use the inter-switch coordination to deliver essential packet processing information between switches. However, only a few of them consider the overhead of such coordination. OmniMon [2] piggybacks its measurement results on packets to synchronize its components. RedPlane [37] replicates the states of in-network functions among switches. ApproSync [29] appends register values to packets to enable low-latency state transmission between the data plane and the control plane. However, none of the above solutions takes the per-packet byte overhead into consideration, inevitably leading to non-trivial degradation in end-to-end performance. Unlike the above solutions, PINT [38] bounds the per-packet byte overhead to a limited user-specified value using probabilistic techniques, which is complementary to Hermes. However, its approach targets the INT scenario, while Hermes can be applied to other scenarios such as software-defined measurement.

Program deployment. Existing program deployment frameworks exploit ILP models or heuristics to deploy programs on programmable switches. However, none of them considers the per-packet byte overhead. More precisely, existing solutions can be classified into two categories. The first category consists of single-switch frameworks, including Min-Stage [8] and Sonata [4]. These frameworks focus on optimizing the deployment of a single program on a single switch. Thus, they inherently fail to optimize the inter-switch coordination. Moreover, the second category includes network-wide frameworks, including SRA [9], SPEED [6], MTP [57], Flightplan [7], Lyra [5], and P4All [59]. They overcome the drawbacks of single-switch frameworks via their capability of network-wide deployment. Nevertheless, they overlook the overhead of the inter-switch coordination. In contrast, Hermes surpasses the above solutions by optimizing the per-packet byte overhead.

VIII. Conclusion

In this paper, we propose Hermes, a framework that aims to optimize the per-packet byte overhead in inter-switch coordination. Our key idea is to integrate the objective of minimizing the overhead into network-wide program deployment. We have implemented Hermes on 32×100 Gbps Tofino-based switches. Our testbed experiments and large-scale simulation indicate that Hermes outperforms existing solutions with lower per-packet byte overheads and higher end-to-end performance.

ACKNOWLEDGEMENT

We thank our reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (2018YFB1800601), the Key R&D Program of Zhejiang Province (2022C01085, 2021C01036), the Science and Technology Development Funds of China (2021ZY1025), the Joint Funds of the National Natural Science Foundation of China (U20A20179), and the National Natural Science Foundation of China (62172007).

REFERENCE

[1] Y. Li, R. Miao, H. H. Liu *et al.*, "Hpcc: high precision congestion control," in *ACM SIGCOMM*, 2019, pp. 44–58.
[2] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *ACM SIGCOMM*, 2020, pp. 404–421.
[3] V. Sivaraman, S. Narayana, O. Rottenstreich *et al.*, "Heavy-hitter detection entirely in the data plane," in *ACM SOSR*, 2017, pp. 164–176.
[4] A. Gupta, R. Harrison, M. Canini *et al.*, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, 2018, pp. 357–371.
[5] J. Gao, E. Zhai, H. H. Liu *et al.*, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *ACM SIGCOMM*, 2020, pp. 435–450.
[6] X. Chen, H. Liu *et al.*, "Speed: Resource-efficient and high-performance deployment for data plane programs," in *IEEE ICNP*, 2020, pp. 1–12.
[7] N. Sultana, J. Sonchack *et al.*, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *USENIX NSDI*, 2021, pp. 571–592.
[8] L. Jose, L. Yan, G. Varghese *et al.*, "Compiling packet programs to reconfigurable switches." in *USENIX NSDI*, 2015, pp. 103–115.
[9] H. Liu *et al.*, "Sra: Switch resource aggregation for application offloading in programmable networks," in *IEEE GLOBECOM*, 2020, pp. 1–6.
[10] Tofino, https://www.barefootnetworks.com/technology/#tofino.
[11] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[12] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX NSDI*, 2013, pp. 29–42.
[13] Q. Huang and P. P. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *IEEE INFOCOM*, 2014, pp. 1420–1428.
[14] M. Moshref *et al.*, "Dream: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM*, 2014, pp. 419–430.
[15] ——, "Scream: Sketch resource allocation for software-defined measurement," in *ACM CoNEXT*, 2015, pp. 1–14.
[16] L. Tang, Q. Huang, and P. P. Lee, "Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *IEEE INFOCOM*, 2019, pp. 2026–2034.
[17] V. Jeyakumar, M. Alizadeh, Y. Geng *et al.*, "Millions of little minions: Using packets for low latency network programming and visibility," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 3–14, 2014.
[18] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling path queries," in *USENIX NSDI*, 2016, pp. 207–222.
[19] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *USENIX NSDI*, 2016, pp. 233–248.
[20] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *USENIX NSDI*, 2016, pp. 311–324.
[21] P. Tammana, R. Agarwal *et al.*, "Distributed network monitoring and debugging with switchpointer," in *USENIX NSDI*, 2018, pp. 453–456.
[22] B. Turkovic, F. Kuipers *et al.*, "Fast network congestion detection and avoidance using p4," in *NEAT*, 2018, pp. 45–51.
[23] B. Turkovic and F. Kuipers, "P4air: Increasing fairness among competing congestion control algorithms," in *IEEE ICNP*, 2020, pp. 1–12.
[24] P. Taheri, D. Menikkumbura *et al.*, "Rocc: robust congestion control for rdma," in *ACM CoNEXT*, 2020, pp. 17–30.
[25] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *ACM SIGCOMM*, 2017, pp. 85–98.
[26] X. Chen *et al.*, "Measuring tcp round-trip time in the data plane," in *SPIN*, 2020, pp. 35–41.
[27] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *USENIX ATC*, 2018, pp. 823–835.
[28] J. Sonchack *et al.*, "Turboflow: Information rich flow record generation on commodity switches," in *ACM EuroSys*, 2018, p. 11.
[29] X. Chen *et al.*, "Approsync: approximate state synchronization for programmable networks," in *IEEE ICNP*, 2020, pp. 1–12.
[30] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[31] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM SIGCOMM*, 2018, pp. 561–575.
[32] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *ACM SIGCOMM*, 2016, pp. 101–114.
[33] Q. Huang, S. Sheng, X. Chen *et al.*, "Toward nearly-zero-error sketching via compressive sensing," in *USENIX NSDI*, 2021, pp. 1027–1044.
[34] C. Kim *et al.*, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM Poster*, 2015.
[35] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, 2017, pp. 121–136.
[36] X. Jin, X. Li *et al.*, "Netchain: Scale-free sub-rtt coordination," in *USENIX NSDI*, 2018, pp. 35–49.
[37] D. Kim *et al.*, "Redplane: enabling fault-tolerant stateful in-switch applications," in *ACM SIGCOMM*, 2021, pp. 223–244.
[38] R. Ben Basat *et al.*, "Pint: Probabilistic in-band network telemetry," in *ACM SIGCOMM*, 2020, pp. 662–680.
[39] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.
[40] A. Roy *et al.*, "Inside the social network's (datacenter) network," in *ACM SIGCOMM*, 2015, pp. 123–137.
[41] P4 Language Consortium. P4C, https://github.com/p4lang/p4c.
[42] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM*, 2016, pp. 511–524.
[43] Z. Meng *et al.*, "Microf: An efficient framework for enabling modularized service chains in nfv," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 8, pp. 1851–1865, 2019.
[44] L. Liao and V. C. Leung, "Lldp based link latency monitoring in software defined networks," in *IEEE CNSM*, 2016, pp. 330–335.
[45] L. Yang, Z.-P. Cai, and H. Xu, "Llmp: exploiting lldp for latency measurement in software-defined data center networks," *Journal of Computer Science and Technology*, vol. 33, no. 2, pp. 277–285, 2018.
[46] V. Chankong and Y. Y. Haimes, *Multiobjective decision making: theory and methodology*. Courier Dover Publications, 2008.
[47] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.
[48] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *ACM SIGCOMM*, 2017, pp. 43–56.
[49] S. Chole *et al.*, "drmt: Disaggregated programmable switching," in *ACM SIGCOMM*, 2017, pp. 1–14.
[50] E. C. man Jr, M. Garey, and D. Johnson, "Approximation algorithms for bin packing: A survey," *Approximation algorithms for NP-hard problems*, pp. 46–93, 1996.
[51] Gurobi Optimizer, http://www.gurobi.com.
[52] PktGen, https://pktgen-dpdk.readthedocs.io/.
[53] Intel Data Plane Development Kit, http://dpdk.org.
[54] S. Knight *et al.*, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.
[55] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *ACM SIGCOMM*, 2018, pp. 576–590.
[56] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown *et al.*, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM*, 2013, pp. 99–110.
[57] X. Chen *et al.*, "Mtp: Avoiding control plane overload with measurement task placement," in *IEEE INFOCOM*, 2021, pp. 1–10.
[58] switch.p4, https://github.com/p4lang/switch.
[59] M. Hogan *et al.*, "Modular switch programming under resource constraints," in *USENIX NSDI*, 2022, pp. 1–15.