Troubleshooting Programmable Data Planes via Real-Time Table Information Recording

Chengyuan Huang, *Member, IEEE*, Yibo Xiao, Tianfan Zhang, Chao Yang, Dong Zhang, Bingheng Yan, Ahmed M. Abdelmoniem, *Senior Member, IEEE*, Gianni Antichi, Xiaoliang Wang, *Member, IEEE*, Fu Xiao, *Senior Member, IEEE*, Wanchun Dou, Guihai Chen, *Fellow, IEEE*, Hao Yin, and Chen Tian, *Senior Member, IEEE*

***Abstract*— While the flexibility of programmable switches brings opportunities, it also introduces security risks. Hence, it is vital to conduct effective troubleshooting in the programmable switch to mitigate frequent network failures. However, troubleshooting programmable switch failures is challenging due to their enhanced flexibility and functionality compared to regular switches, posing increased difficulty in debugging, particularly with limited debugging tools and information. To address this problem, we propose an efficient troubleshooting method that records real-time information about packets in the data plane, including the tables involved in packet processing. Unfortunately, due to hardware limitations, it is infeasible to record all tables' information in the data plane. Thus, the key is to find the table set reflecting the execution path a packet goes through while minimizing the resource overhead. We first represent P4 programs as a probabilistic transition directed acyclic graph (DAG) and employ information entropy to quantify the information within a set of tracked tables. Then, we adopt a two-step approach and design algorithms to find both optimal and approximately optimal table record plans. The evaluation results show the efficacy of the proposed method, including achieving the same path recovery rate as the related works with less than one-third of the resource consumption.**

***Index Terms*— Programmable data plane, network troubleshooting.**

Received 23 April 2024; revised 1 December 2024; accepted 9 February 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor K. Park. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2901502, in part by the Key Program of Natural Science Foundation of Jiangsu under Grant BK20243053, in part by the National Natural Science Foundation of China under Grant 62325205 and Grant 62172204, in part by the Taishan Industrial Experts Program under Grant tscx202312005, and in part by Nanjing University-China Mobile Communications Group Company Ltd. Joint Institute. *(Corresponding author: Chen Tian.)*

Chengyuan Huang, Yibo Xiao, Tianfan Zhang, Chao Yang, Xiaoliang Wang, Wanchun Dou, Guihai Chen, and Chen Tian are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: ryanhuang1014@gmail.com; 201220094@smail.nju.edu.cn; zhangtf19@gmail.com; cyang@smail.nju.edu.cn; waxili@nju.edu.cn; douwc@nju.edu.cn; gchen@nju.edu.cn; tianchen@nju.edu.cn).

Dong Zhang and Bingheng Yan are with Jinan Inspur Data Company Ltd., Jinan 250101, China (e-mail: zhangdong@inspur.com; yanbh@inspur.com).

Ahmed M. Abdelmoniem is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K. (e-mail: ahmed.sayed@qmul.ac.uk).

Gianni Antichi is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K., and also with the Dipartimento Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milan, Italy (e-mail: g.antichi@qmul.ac.uk).

Fu Xiao is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (e-mail: xiaof@njupt.edu.cn).

Hao Yin is with the Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China (e-mail: h-yin@mail.tsinghua.edu.cn).

Digital Object Identifier 10.1109/TON.2025.3541884

I. INTRODUCTION

PROGRAMMABLE data planes have emerged as an alternative to the traditional data plane, whose functionalities were hard-coded. Network applications based on programmable switches have sprung up, such as network measurement [1], [2], [3], in-network computing [4], and in-network storage [5], [6]. Programmable switches are also deployed on a large scale in production networks as cloud gateways [7], [8], [9].

However, these in-network applications can also bring new problems to the packet processing pipeline in the switches. The problems can be caused by program bugs in the data plane or control plane, compiler bugs, hardware errors, etc. While existing network troubleshooting is already challenging, the complexity is heightened with programmable switches. Unlike regular switches, programmable switches offer extensive data plane functionalities that were traditionally handled by a general-purpose CPU. However, constrained hardware resources prevent programmable switches from generating comprehensive logs and debugging information akin to CPUs, resulting in insufficient data to pinpoint network issues.

Some existing works [10], [11], [12], related to network troubleshooting, tend to mirror packets from switches to CPUs and then analyze the packets to find the root cause. However, these works focus on traditional network errors such as routing errors, ACL misconfigurations, etc., and do not consider errors originating from new functionalities introduced by programmable switches. For example, in Sailfish [7], some network functions such as DDoS, firewall, and NAT are offloaded to the programmable switches. The failure of these programmable switches' new complex functions is beyond the capability of existing network troubleshooting systems.

To tame the complexity in the data plane, it is essential to track the complete behaviors of packets within the switch for troubleshooting. Snapshot is a useful tool provided by commercial switches like Tofino [13]. It can capture a network packet in programmable switches and produce useful information about the packet, such as state data and behavior of the packet in the switch. However, Snapshot has key limitations (elaborated in Section II-B), e.g., it cannot produce debugging information at runtime. Consequently, Snapshot is suitable for debugging when a network failure is reproducible, but fails to handle many unreproducible network failures [14].

To monitor the complete processing behavior of packets in the data plane, the real-time table information should be recorded for each packet at runtime. This provides the fine-grained processing information for troubleshooting. Using this information, we can replay each packet's processing to detect errors effectively. For example, consider a P4 program that specifies a packet must pass through Table A before reaching Table B. However, due to a compiler error—common in practice [15]—Table A and Table B are mistakenly placed in different conditional branches, resulting in packets that hit Table A never reaching Table B. In such cases, existing troubleshooting tools require significant effort to determine which table caused the error. In contrast, by recording real-time table information for each packet during runtime, we can easily pinpoint that the issue arises from the incorrect placement of Table B, thereby identifying the specific compiler error. Specifically, the recorded table information includes *execution* and *hit* information. The *execution* information refers to which tables the packet goes through, while the *hit* information refers to whether these tables are hit or not. This information is stored in Packet Header Vectors (PHVs) and is appended to the packet header when the packet is mirrored. Examining the real-time table information in the mirrored packet can provide accurate and comprehensive information for the execution path, significantly facilitating troubleshooting.

Unfortunately, it is impractical to record information for all tables due to hardware constraints in programmable switches regarding PHV and stage resources [16], [17], [18]. On the one hand, PHVs are used to record table information; however, due to their limited capacity, e.g., less than 80 bits are available after enabling the basic functionalities [11], the number of tables that can be tracked is constrained. On the other hand, stages serve as action units in the switch pipeline, and a table must be placed in one of the stages. However, recording table information may introduce a write-write dependency between tables, preventing them from sharing the same stage and thus increasing the required stages in the P4 program. Given the limited number of stages in a programmable switch, e.g., 12 stages in the Tofino chip [13], the dependency between tables exacerbates the stage scarcity problem, resulting in fewer trackable tables. Hence, with a real P4 program deployment that may contain over a hundred tables inside, e.g., the basic switch functionality `switch.p4`, recording every table information is infeasible. Therefore, designing an effective table record plan that covers the crucial execution path and maximizes table information collection within hardware constraints poses a great challenge. *To the best of our knowledge, no prior work has adequately solved this problem.*

To tackle it, we first introduce the entropy for assessing information in tracked tables and use the entropy to guide us to derive a table record plan that induces efficient coverage containing more information. Specifically, in information theory [19], the entropy represents the average level of information and can be used to evaluate the information within a set of tables. The insight is that a table set with higher informativeness indicates a wider range of possibilities, necessitating the need to record these tables' information. For example, if we can find a minimal table set with the highest informativeness, we can monitor the complete execution path for more packets by examining the packets' header, with the minimal number of tables. Hence, we ingeniously transform the original problem of finding an efficient table record plan that tracks the execution path of more packets into identifying a table set with higher information entropy within hardware constraints.

Later, to apply information entropy, we introduce probability into the procedure of the table set finding. Therefore, we first model the execution of a P4 program as a probabilistic transition direct acyclic graph (DAG). Then, we design a depth-first search (DFS)-based algorithm to efficiently compute the information entropy from a set of tracked tables. The objective of leveraging the information entropy as a metric is to find a table record plan with the largest information entropy while meeting the switch hardware constraints. It is a complex combinatorial optimization problem. To solve it, we adopt a two-step approach, first generating a set of candidate tables according to a certain strategy, and then using a Satisfiability Modulo Theories (SMT) solver to determine whether it satisfies the constraints. We design two strategies, *Branch-and-Bound (BnB)* and *Greedy* Algorithms, to generate a candidate table set and obtain the *optimal* and *approximately optimal* table record plans, respectively.

To simplify the record plan computation, we first consider only the table *execution* information when introducing our information entropy model and algorithms in Section IV. We then extend the model to include the table *hit* information and revise the algorithm to solve the extended problem in Section V. Our contributions are as follows:

1) We record real-time table information in the programmable data plane for troubleshooting, while considering the limitations of available resources.
2) We first leverage the information entropy as the metric to select recorded tables and design a DFS-based algorithm to calculate the entropy efficiently.
3) We propose a two-step solution using *Branch-and-Bound* and *Greedy* algorithms to find the *optimal* and *approximately optimal* table record plans, respectively.
4) We evaluate the proposed approach using real P4 programs and demonstrate its effectiveness in achieving low-overhead troubleshooting of programmable data planes.

II. Background and Motivation

A. Programmable Switch

Programmable switches provide engineers with the flexibility to customize their packet processing logic in languages like P4 [20]. Engineers can specify the parser which extracts headers of a packet into metadata and match+action tables in the data plane. A packet passes through different tables and if it matches an entry of each table, the corresponding action is executed. Branch conditions are supported in P4 so that packets may go through different processing orders. The P4 program can be modeled as a DAG, in which nodes represent the tables or statements, and edges represent the execution order.

From the perspective of hardware architecture, programmable switches consist of several Match-Action Unit (MAU) stages. All tables in a P4 program are placed into these MAU stages. Each MAU stage has its isolated hardware resources, such as SRAM, TCAM, Stateful ALU, etc. Multiple tables can be placed in the same stage to achieve parallelization without violating the hardware constraints. The switch also has PHV resources, which are used to store some fields extracted by the parser, metadata, and some temporary variables.

B. *Debugging Programmable Switch*

Snapshot is a useful debugging tool provided by Tofino switches [13]. Engineers can set a trigger in a pipeline and it will capture the first network packet that satisfies the trigger condition. Snapshot will show all the information about this packet at each stage, i.e., all the fields of the PHV, the executed tables, and the matched actions. This information can effectively help engineers to locate problems in P4 programs.

Unfortunately, Snapshot has some limitations. First of all, Snapshot can only capture the first network packet that meets the conditions, so it may miss the following packets of interest. Secondly, Snapshot can only capture the information of a network packet in one pipeline at a time. This can be problematic since Tofino switches have two pipelines (i.e., Ingress and Egress) and the number of pipelines can be 4 or more for P4 programs that use pipeline folding technology [7]. Thirdly, the setting of triggers has its limitations. Based on our experience with Tofino switches, we find that only PHV fields that appear in the branch condition or the key of a table can be used as the trigger condition. Therefore, there may be a situation where a packet passes through the switch, but Snapshot cannot capture the packet. Finally, Snapshot is an offline debugging tool similar to the GNU Project Debugger (GDB), which does not provide real-time feedback for every network packet. While Snapshot is effective for troubleshooting reproducible network failures and can offer valuable insights, not all network failures are reproducible. Many network issues may temporarily resolve after a reboot [14], making Snapshot less useful in such cases.

C. *Real-Time Recording in Packet Header*

The switch programmability makes it possible to record table information of network packets at runtime. For example, if we want to know whether the packet has passed a certain table, we can assign a variable *path*, and then add a line of code to the table's actions, which sets one bit of *path* to 1. Then, it is easy to determine whether the packet has passed through the table by examining the value of the *path* variable. To communicate the *path* value outside the switch, we can mirror the packet to a specific port and append the *path* value to the packet's header. By parsing the appended headers of the mirrored packet, the packet's execution path within the switch can be recorded.

However, due to the hardware limitations of programmable switches, it is infeasible to record information about all tables. The hardware bottleneck mainly results from the scarcity of the PHV and stages. 1) The PHV scarcity. If we record all tables' execution information, then one bit should be assigned to each table. As a result, for a basic forwarding program that has O(100) tables, at least O(100) PHV bits are needed for recording. If the switch performs more functions or records finer-grained information, more PHVs are required, e.g., if the table hit information is recorded, 2x PHV bits will be consumed. However, PHVs are very scarce resources, e.g., the total PHV capacity of the Tofino chip is only 4K bits. The implemented functionalities consume the majority of the PHVs. For instance, after implementing the basic switch functions (`switch.p4`) and the flow event telemetry (netseer [11]), the PHVs consumption already exceeds 98% and the available PHVs is less than 80 bits. Consequently, given the common case where over a hundred tables are in the data plane, assigning a PHV bit for each table for recording is infeasible.

2) The stage scarcity. The number of available stages also limits the table record plan. This is because the placement of tables into stages is primarily constrained by table dependencies. Only independent tables that can be parallelized can be placed in the same stage. However, PHVs consist of 8-bit, 16-bit, and 32-bit containers, indicating that multiple tables may share a single PHV container to record their information. This introduces write-write dependencies [8] between these tables, preventing them from being placed in the same stage. As a result, more stages are required to accommodate the same number of tables. Given the limited number of stages (e.g., 12 stages in Tofino [13]) and the large number of deployed tables (e.g., O(100) tables), finding a feasible table placement within the stage constraint becomes a significant challenge.

III. Information Entropy

Because of the limitations introduced in Section II-C, not all the information can be recorded in the data plane, and so we have to use limited resources to record as much information as possible. To find the optimal solution for table information collection, a metric to evaluate the various table record plans is needed. This section introduces the concept of information entropy and explains how it can be applied to the scenario of table information collection.

Given a P4 program, which consists of n tables and PHV containers, which are available for collecting table information. PHV containers are classed into groups of 8-bit, 16-bit, or 32-bit, to which we assign the tables without violating the constraints. Technically, more bits can be allocated to a table for collecting more information. The states of a table can be divided into three types, *not executed*, *executed but not hit*, and *hit*, which represent that the packet does not pass through the table, passes through the table but does not hit any entry, and hits an entry, respectively. We refer to whether a table is executed as *execution information* and whether a packet hits an entry in the table as *hit information*. Allocating 1 bit can record the table execution information, and allocating 2 bits can record all three states, i.e., both the execution information and hit information are recorded. Allocating more bits can record more detailed information, such as which action is matched and executed. In order to simplify the problem, we will first assign only one bit to a

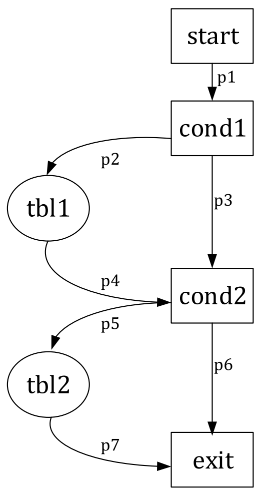


Fig. 1. Probabilistic transition DAG.

table to record table execution information. We consider the table hit information in Section V.

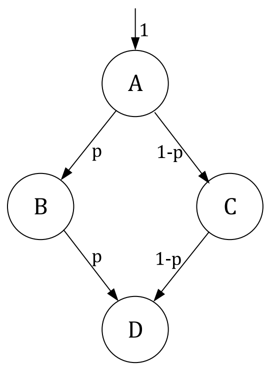
In Section III-A, we reinterpret a P4 program as a probability transition DAG; Section III-B introduces how we use information entropy as a metric to evaluate the table execution information; Section III-C introduces an efficient algorithm to compute information entropy.

A. *Probabilistic Transition DAG*

As introduced in Section II-A, each P4 program can be viewed as a DAG. Whenever a packet passes through the processing pipeline, it will take different branches and may hit different match-actions according to the value of its header and the match-action entries in tables. From the perspective of data streams, the P4 program can be regarded as a probabilistic transition DAG. Fig. 1 is an example of a probabilistic transition DAG. The branch selection and table matching of each packet in the stream can be viewed as probabilistic behavior. The round node represents a *table* in the P4 program into which we can inject code to record table information. The square node represents an *if* condition statement in the P4 program into which we cannot add code for recording table information. Each directed edge has a weight that represents the probability of transitioning from one node to another node. For instance, in the example of Fig. 1, when a packet in the data stream passes through the *cond1* node, it will either pass through the *tbl1* table with probability p_2 or arrive directly at the *cond2* statement with probability p_3.

Since all packets entering a table will exit the table, the sum of all incoming edge probabilities and all outgoing edge probabilities of any node will always be equal, i.e., $p_1 = p_2 + p_3, p_2 = p_4, p_3 + p_4 = p_5 + p_6, p_5 = p_7$. Each DAG has a start node and an exit node, and their probability is 1, i.e., $p_1 = 1, p_6 + p_7 = 1$.

We can use P4 *counter* primitive [20] to count the selection of different branches and the hits of each table. The counters can be used to measure the frequency of the estimation of the probability. Unfortunately, programmable data plane resources are limited, and assigning a *counter* to each table is impractical. To overcome this issue, the data stream can



Recorded Table(s)	Entropy
A	0
B	-plog₂p-(1-p)log₂(1-p)
C	-plog₂p-(1-p)log₂(1-p)
D	0
B, C	-plog₂p-(1-p)log₂(1-p)

Fig. 2. A simple DAG and its entropy.

be sampled and then replayed on the P4 software switch, e.g., BMv2 [21] or Tofino's ASIC emulator, to estimate the probability associated with all the tables.

B. *The Entropy Metric*

Because of hardware limitations, it is not possible to record all the table execution information. However, we observe that it is unnecessary to record all tables. Specifically, we observe that not all tables contain useful information, and some tables contain repeated information. The left part of Fig. 2 shows a simple probabilistic transition DAG. It shows that Table A and Table D are bound to be executed, so their probability is 1, and they do not contain the execution information from an entropy perspective. The execution information of table B and table C overlaps with each other because these two tables are complementary, i.e., when table B is not executed, that means table C must be executed. Therefore, in order to obtain all the table execution information of the DAG, we only need to record either table B or table C but not all four tables.

Therefore, to determine which tables to record, we need to measure the information contained in different table record plans. In this work, we use information entropy [19] as a metric to measure the information of table record plans. The execution of a table can be regarded as a binary (0 or 1) random variable. So, a plan for table information collection can be considered as a set of multiple random variables corresponding to each table. Then, the joint entropy of these random variables can be regarded as the entropy metric of the plan. Fig. 2 demonstrates a simple DAG and the entropy of some recorded tables. For example, the execution state of table A is represented as a random variable a where 1 means executed and 0 means not executed. The information entropy of a is

$$\Sigma_a p(a) log_2(1/p(a)) = 1 * log_2(1) + 0 = 0$$

Similarly, we can calculate the information entropy of recording table B, C and D. The entropy of recording both B and C is the joint entropy of b and c, i.e.,

$$\Sigma_b \Sigma_c p(b,c) log_2(1/p(b,c)) = -p log_2(p) - (1-p) log_2(1-p)$$

We find that information entropy is a reasonable metric, and its value is consistent with our previous observations. That is, table A and table D do not contain information, and the information in table B and table C overlaps with each other.

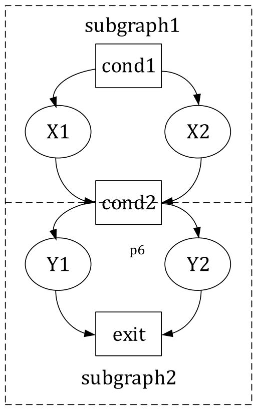


Fig. 3. Subgraphs of DAG.

C. How to Compute Entropy

The interaction relationship of multiple random variables is too complex, so this paper only considers the correlation that can be derived from the program structure, which means that some tables are mutually exclusive and cannot both be executed, while some other tables are coexistent and the execution or hit of one table necessarily leads to the execution of the other. Additionally, we assume that the execution and hits of different tables are independent. For example, in Fig. 3, there are 4 random variables, *X1*, *X2*, *Y1*, and *Y2*, which respectively indicate whether the table in which they are located is executed. According to our assumption, *X1* and *X2* are not independent of each other because they are mutually exclusive. But the two random variable groups (*X1*, *X2*) and (*Y1*, *Y2*) are independent of each other, so $P(X_1, X_2, Y_1, Y_2) = P(X_1, X_2)P(Y_1, Y_2)$ and $P(X_1, Y_1) = P(X_1)P(Y_1)$.

Even though the example may seem simple, computing the entropy of multiple random variables is still non-trivial. Suppose we record *n* tables of the P4 program, a strawman approach to obtain record plans is to enumerate all possibilities and compute their probabilities, which has an unacceptable time complexity of $O(2^n)$. We observe that many combinations of random variables are infeasible because the P4 program is a DAG. As per the previous example in Fig. 3, X_1 and X_2 were both 0, which is an impossible combination because the packet will definitely choose either one of the tables represented by the random variables X_1 or X_2.

To address this problem, we design a depth-first search (DFS)-based algorithm that can efficiently compute the information entropy. We first observe that the DAG has many nodes that a packet will definitely pass through, such as *cond1* and *cond2* in Fig. 3, which we name the *cuts*. The *cuts* divide the DAG into several subgraphs, such as *subgraph1* and *subgraph2* in Fig. 3. Based on the assumption of mutual independence, the groups of random variables of each subgraph are independent of each other, and so the entropy of the whole DAG is the sum of the entropy of each subgraph. We can perform a depth-first search on each subgraph to calculate its information entropy and then sum them up to get the total information entropy.

Algorithm 1 describes how to calculate the information entropy of the whole graph and a subgraph. The input of

Algorithm 1 Compute Entropy

```
Input: DAG, $p_e[][]$, $p_n[]$, $recorded\_set$
Output: Entropy
1:  Function $calculate\_total\_entropy()$ do
2:      $entropy \leftarrow 0$
3:      Split DAG into subgraphs Array $G_{sub}[1 \ldots n]$
4:      for $i = 0 \rightarrow n-1$ do
5:          $node \leftarrow$ the first node in subgraph $G_{sub}[i]$
6:          $recorded \leftarrow empty$
7:          $path\_set \leftarrow empty$
8:          $subg\_entropy \leftarrow 0$
9:          $DFS(G_{sub}[i], node, 1, recorded, path\_set)$
10:         for $recorded, p$ in $path\_set$ do
11:             $subg\_entropy \leftarrow subg\_entropy - p * log_2(p)$
12:         end for
13:         $entropy \leftarrow entropy + subg\_entropy$
14:     end for
15:     return $entropy$
16: end Function
17:
18: Function $DFS(G, node, p, recorded, path\_set)$ do
19:     if $node$ is the final node in $G$ then
20:         if $recorded$ not in $path\_set$ then
21:             insert $(recorded, p)$ into $path\_set$
22:         else
23:             $path\_set[recorded] += p$
24:         end if
25:         return
26:     end if
27:     if $node$ in $recorded\_set$ then
28:         append $node$ into $recorded$
29:     end if
30:     for all neighbor $neigh$ of $node$ in $G$ do
31:         $DFS(G, neigh, p\frac{p_e[node][neigh]}{p_n[node]}, recorded, path\_set)$
32:     end for
33: end Function
```

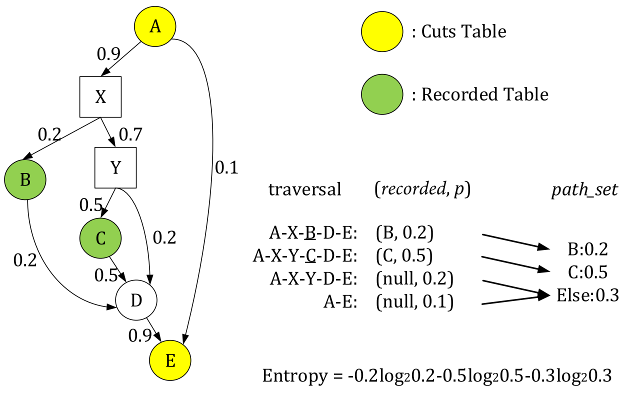


Fig. 4. Calculate subgraph entropy.

Algorithm 1 is the probabilistic transition DAG of a program (DAG), the probability of each edge ($p_e[][]$), the probability of each node ($p_n[]$), and the table record plan ($recorded\_set$). The output of Algorithm 1 is the entropy of this DAG (Entropy). The $calculate\_total\_entropy$ in Algorithm 1 first initializes $entropy$ to 0 and splits DAG into n subgraphs $G_{sub}[1..n]$ (lines 2 - 3). For each subgraph i, we initialize the recorded table set ($recorded$), the traversed execution path set ($path\_set$), and the entropy of the subgraph ($subg\_entropy$) to empty, empty, and 0, respectively (lines 5 - 8). Later, the *DFS* calculates the information entropy of a subgraph through a depth-first traversal (line 9). After traversing all paths, we traverse $path\_set$ and accumulate each path's entropy to obtain the subgraph's entropy (lines 10 - 12). Finally, the entropy values of all subgraphs are accumulated to derive the whole entropy of the graph (line 13).

The DFS algorithm traverses all paths in the subgraph and computes the execution probability of each path. For each path, $recorded$ is used to include all nodes specified in $recorded\_set$ that need to be recorded along that path (lines 27-29). Note that the execution of one path corresponds to the execution situation of all recorded nodes. For a specified path, if it is executed, it indicates that recorded nodes in $recorded$ are executed and those not in $recorded$ are not executed. So as long as we traverse all possible execution paths, we enumerate all potential valid combinations of recorded table execution situations. Note that different paths may have the same $recorded$ list, which means that these paths correspond to the same execution situation of the recorded nodes. So, we accumulate the probability of these paths (lines 20 - 23). The pair ($recorded$, p) in $path\_set$ represents the probability p of the event where the nodes in the set $recorded$ are executed, while the nodes outside this set remain unexecuted. Thus, $path\_set$ contains all possible execution situations of the recorded table and its probability.

Fig. 4 gives an example of calculating the entropy of a subgraph, given the recorded table plan, i.e., $recorded\_set = (B, C)$. This subgraph consists of 5 tables A, B, C, D, E and 2 conditional statements X, Y. Table A and E are *cuts* tables, whose probability is 1, and the recorded tables are B and C. Algorithm 1 will traverse all paths: (1) A-X-B-D-E with recorded table B. (2) A-X-Y-C-D-E with recorded table C. (3) A-X-Y-D-E with recorded table $null$. (4) A-E with recorded table $null$. Note that (3) and (4) have the same $recorded$ so we sum up their probability and get the $path\_set$ in Fig 4. Then we can calculate the subgraph entropy as $-0.2log_2 0.2 - 0.5log_2 0.5 - 0.3log_2 0.3$.

IV. The Optimal Table Record Plan

This section discusses how to find the optimal table record plan. First, we give a formal description of the problem. Given a P4 program consisting of n tables and m PHVs available for recording table information, we assign some bits of PHVs to the recorded tables to store whether the table is executed or not. An assignment can be formally represented by an $n * m$ matrix A. $A_{ij} = 1$ means that we assign 1 bit of PHV_j to $table_i$. For a P4 program consisting of 3 tables and II available PHVs, if we assign 1 bit of PHV_1 to $table_2$ and 1 bit of PHV_2 to $table_1$, the assignment matrix will be like:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \quad (1)$$

As introduced in Section II-C, the hardware limitations are mainly due to limited **PHV** and **stage**. PHV is organized in the form of a container, and the container has only widths of 8bit, 16bit, and 32bit. Therefore, the widths of these m available PHVs are $\{w_1, w_2, \ldots, w_m\}$, where $w_i = 8$ or 16 or 32. The restriction of PHV can be formally described as $\Sigma_i A_{ij} \le w_j$.

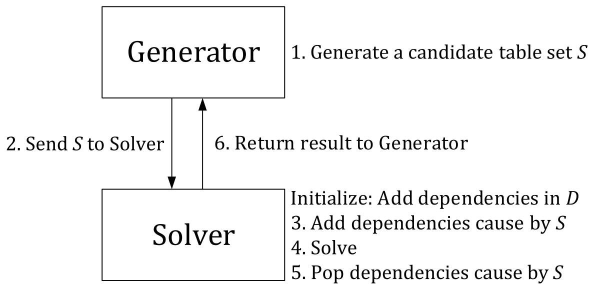

Generator
1. Generate a candidate table set S
2. Send S to Solver
6. Return result to Generator
Solver
Initialize: Add dependencies in D
3. Add dependencies cause by S
4. Solve
5. Pop dependencies cause by S

Fig. 5. Workflow of two-step solving method.

The stage restriction is more complex to deal with. We use $stage_i$ variable to denote the stage where $table_i$ is placed. Each P4 program has complex table dependencies, which we roughly divide into two categories. The first one is that $table_i$ must be placed before $table_j$, i.e., $stage_i < stage_j$; the other one is that $table_i$ cannot be placed after $table_j$, i.e., $stage_i \le stage_j$. We denote the table dependencies in the original P4 program, which is prior to the injection of table information collection code, as D.

This may cause new write-write dependencies when a PHV is assigned to multiple tables. Specifically, if $table_i$ and $table_j$ are not mutually exclusive, then assigning the same PHV to them will create a write-write dependency, and so these tables cannot be placed in the same stage. Two tables are considered mutually exclusive if it is logically impossible for both to be executed simultaneously. For example, in Fig. 4, Table *B* and *C* are mutually exclusive because for a packet, only one of Table *B* and *C* may be executed.

After introducing the notations and restrictions, the problem can be formally modeled as follows: our objective is to find an optimal assignment matrix with maximum information entropy within the hardware constraints.

$$\begin{aligned} &\underset{A}{\operatorname{argmax}}\, Entropy(A) \\ \text{subject to } &\Sigma_i A_{ij} \le w_j \\ &\text{Dependencies in } D \\ &\text{Dependencies caused by } A \end{aligned}$$

A. A Two-Step Solving Method

The problem described above is a complex combinatorial optimization challenge, with a non-linear objective function and constraints that are difficult to express mathematically. To solve this problem, we take a two-step approach. In the first step, we generate a candidate assignment, and in the second step, we verify whether the candidate assignment satisfies the constraints.

Fig. 5 shows the two-step method's workflow, which consists of two modules, **Generator** and **Solver**. The Generator creates a candidate table set S according to the strategies in Section IV-B or Section IV-C (step 1) and sends it to the Solver (step 2). S is a set of recorded tables different from the existing assignment A. The Solver determines if there is an assignment such that all tables in S are recorded and all constraints are satisfied (steps 3 - 5). Later, the result is returned to the Generator (step 6). The Generator produces the next candidate table set after the result returned by the Solver (step 1 again).

TABLE I

NOTATION AND CONSTRAINT IN SOLVER

Notation	
Table Set:	$table_{s_1}, table_{s_2}, ..., table_{s_n}$
Stage:	$stage_{s_1}, stage_{s_2}, ..., stage_{s_n}$
PHV Number:	$PHVnum_{s_1}, PHVnum_{s_2}, ..., PHVnum_{s_n}$
PHV Width:	$w_1, w_2, ..., w_m$
Constraint:	
C1:	for all table, $1 \leq stage_i \leq MaxStage$
C2:	for $1 \leq i \leq n$, $1 \leq PHVnum_{s_i} \leq m$
C3:	for $1 \leq i \leq n, 1 \leq j \leq m$, $count(PHVnum_{s_i}) \leq w_j$, where $PHVnum_{s_i} = j$
C4:	stage constrain of the original program
C5:	$stage_{s_i}! = stage_{s_j}$, if $PHVnum_{s_i} = PHVnum_{s_j}$

This process continues until the optimal solution is obtained. Since D may be large, we add the dependencies of D to the Solver's constraints during initialization and then add (step 3) and pop (step 5) the dependencies related to S incrementally.

Table I shows the notation and constraint of the Solver. *Table Set* is the candidate table set given by the Generator; $stage_i$ represents the stage where $table_i$ is placed; $PHVnum_i$ represents the index of the PHV used to record $table_i$; PHV width w_i indicates the bit width of the PHV. Constraints C1 and C2 limit the range of stage and PHV; Constraint C3 guarantees that PHV usage will not exceed the bit width of each PHV; Constraint C4 is the stage constrain of the original P4 program, which we can get from compiler output; Constraint C5 represents the write-write dependency caused by writing the same PHV.

We use the Z3 SMT Solver [22] to solve the constraints commonly used in constraint problems related to programmable switches [8], [23]. If no solution satisfies the constraints, the Z3 solver returns "unsat" (i.e., unsatisfied). Otherwise, the Z3 solver returns one of the solutions, and we can get a valid PHV array. Based on the candidate table set and PHV, we can get a valid candidate assignment. For example, if the candidate table set $= \{1, 2\}$ and PHV Number$= \{2, 1\}$, we will get an assignment matrix A in equation 1, where we assign one bit of PHV_1 to $table_2$ and one bit of PHV_2 to $table_1$.

B. Branch and Bound

In this section, we introduce the strategy of generating a candidate table set. A strawman or brute force method is to enumerate all valid table sets, but the time complexity is too high to compute the optimal solution for large P4 programs. Hence, we design a Branch-and-Bound (BnB) [24] algorithm to solve this problem, which is a known algorithmic paradigm for solving large-scale NP-hard combinatorial optimization problems.

Algorithm 2 shows the steps to generate the candidate and obtain the optimal solution. The input of Algorithm 2 includes the whole tables determined by the P4 program ($tables[]$), the solver used to determine whether the current plan satisfies all the constraints (Solver), the total width of PHVs (w_{total}), and the tables that are specified and need to be recorded by the user (C). The output of Algorithm 2 includes the maximum entropy ($max\_entropy$) with the optimal table record assignment (A). Initially, C is assigned to the candidate table record set ($candidate$), to ensure the derived table record plan includes C. Next, the algorithm explores the rest of the tables except C (lines 2 - 3). Later, all recordable tables are sorted in descending order according to their information entropy, and all initialization is done (lines 4 - 6). Then, the algorithm searches the sorted tables (line 7) to decide which ones to add to the candidate table set (lines 29 - 32). During the search, we update the current maximum entropy and optimal assignment (lines 16 - 19). We prune the search tree in two cases. First, if a candidate table set is found to be invalid by Solver, then the corresponding branch will be pruned (lines 11 - 15). Second, if the upper bound of the entropy obtained by the search branch is smaller than the current optimal solution, the branch will be pruned (lines 23 - 28).

Algorithm 2 Branch and Bound

```
Input: $tables[]$, Solver, Total width of PHVs $w_{total}$, tables
    that must be covered $C$
Output: Maximum Entropy $max\_entropy$, Optimal Assign-
    ment $A$
1:  Function $branch\_and\_bound()$ do
2:      $candidate \leftarrow C$
3:      $tables = tables - C$
4:      sort the recordable $tables$ array in descending order of
        the entropy
5:      $max\_entropy \leftarrow 0$
6:      $A \leftarrow null$
7:      $search(1)$
8:      return $max\_entropy, A$
9:  end Function
10: Function $search(index)$ do
11:     if $candidate$ is valid (based on Solver) then
12:         get a valid assignment $A\_valid$ from Solver
13:     else
14:         return
15:     end if
16:     $entropy \leftarrow compute\_total\_entropy()$
17:     if $entropy > max\_entropy$ then
18:         $max\_entropy \leftarrow entropy, A \leftarrow A\_valid$
19:     end if
20:     if $w_{total} \leq candidate.size$ then
21:         return
22:     end if
23:     $w_{avail} \leftarrow w_{total} - candidate.size$
24:     $E_{sum} \leftarrow$ sum of first $w_{avail}$ entropy values after $index$
25:     $upper\_bound\_entropy \leftarrow entropy + E_{sum}$
26:     if $upper\_bound\_entropy < max\_entropy$ then
27:         return
28:     end if
29:     insert $tables[index]$ into $candidate$
30:     $search(index + 1)$
31:     remove $tables[index]$ from $candidate$
32:     $search(index + 1)$
33: end Function
```

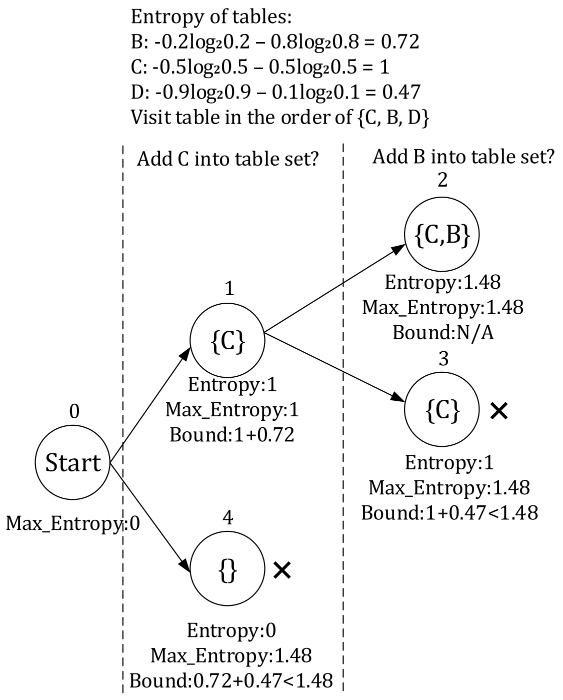


Fig. 6. Example of branch and bound algorithm.

We emphasize the importance of estimating the upper bound of entropy. Assume the sum of PHV bit width is w_{total}, $candidate.size$ is the number of currently selected tables, and w_{avail} is the remaining PHV bit width. Then, the upper bound of the information entropy will not exceed the sum of the current $entropy$ plus the first w_{avail} entropy values after the index of the current table (lines 23 - 25), i.e., the largest w_{avail} entropy values in the $tables$ array that have not been searched yet. This is due to a basic property of information entropy, i.e., $H(X_1, X_2, \ldots, X_n) \leq H(X_1) + H(X_2) + \ldots + H(X_n)$, and the equation holds only when random variables are independent.

Suppose we have a 2-bit PHV container and need to find the optimal table set from the DAG of Fig. 4. Fig. 6 shows the execution process of the BnB algorithm. First, we calculate the information entropy of B, C, and D, respectively, and then sort them. We ignore X, Y because they represent branch statements and are not recordable, and ignore table A, E because they are *cuts*, and their information entropy is 0.

We search through the array $\{C, B, D\}$. Fig. 6 shows the search tree where the numbers above circle nodes indicate the search order. *Node 1* adds table C to the candidate table set, then calculates its information entropy and updates $Max\_Entropy$ and the upper bound. $Max\_Entropy$ is updated from 0 to 1 because the information entropy of $\{C\}$ is 1. The upper bound is $1 + 0.72$ where 1 is the information entropy of $\{C\}$ and 0.72 is the entropy of $\{B\}$ because table B is the first table after C in $\{C, B, D\}$, and it is also the table with the largest entropy among $\{B, D\}$ that have not been searched yet. Next, we present the actions of each node.

Node 2 adds table B to the candidate set, and the current entropy is calculated as 1.48. The information entropy of the candidate table set $\{C, B\}$ is 1.48 instead of 1.72, because table B and table C are not mutually independent. Fig. 4

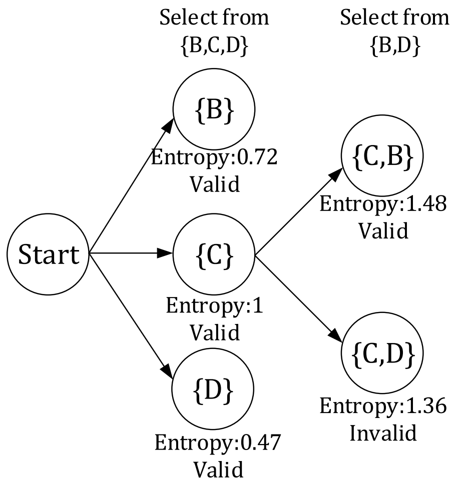


Fig. 7. Example of greedy algorithm.

shows the calculation process. Since two nodes have been selected, the search along this path is complete.

Node 3 does not add table B to the candidate set, so its entropy is still 1 which is less than $Max\_Entropy$, so $Max\_Entropy$ is not updated. The upper bound is $1 + 0.47$ which is less than $Max\_Entropy$, so the branch is pruned.

Node 4 does not add table C to the candidate set. The upper bound is $0.72 + 0.47$, which is the sum of the entropy values of B and D. Hence, the branch is pruned because the upper bound is less than the current $Max\_Entropy$. To elaborate, the information entropy will not exceed the value of $0.72 + 0.47$ even if both B and D are added to the candidate set.

After the search, the algorithm obtains the maximum entropy of 1.47, and the optimal table set is $\{C, B\}$. The assignment method is to assign 1 bit to each of the 2-bit PHV containers to B and C. Since there is only one PHV container in the example of Fig. 6, the assignment is obvious, but if there are multiple PHV containers, the Solver will give a specific valid assignment.

C. Greedy Algorithm

Although BnB can prune some branches and find the optimal solution, the search is very time-consuming for large-scale problems. So, we propose a greedy algorithm that can find an approximate optimal solution in a short time. The main idea of the greedy algorithm is to select the valid table with the largest information entropy from the table array and add it to the candidate table set in each round until w_{total} tables are inserted or no valid table can be found from the array.

We reuse the DAG in Fig. 4 and attempt to find the optimal solution where the PHV width is 2. In addition, we assume that $\{C, D\}$, $\{B, D\}$ is invalid due to the write-write dependencies. Fig. 7 shows the execution process of the greedy algorithm. In the first round, we select from $\{B, C, D\}$, calculate the information entropy of $\{B\}$, $\{C\}$, $\{D\}$, and let the Solver determine whether they are valid or not. This results in C being inserted into the table set. In the second round, we select from $\{B, D\}$, repeat the previous step, and finally choose to add B to the table set. At this point, w_{total} tables are selected, the algorithm terminates, and the solution is $\{C, B\}$.

This greedy algorithm cannot always find the optimal solution. Consider that in Fig 4, we assume that $\{C, D\}$, $\{C, B\}$ is invalid due to the stage constraint. This greedy algorithm will return the value of 1 as the max entropy while the optimal result is 1.16 (The optimal recorded plan is $\{B, D\}$ not $\{C\}$). When a table with larger entropy prevents the selection of several other tables with smaller entropy, whose combined entropy exceeds that of the first table, it results in a suboptimal table record plan. For an n-element table array [A, B, C, ..., N], where all tables have the same entropy e, table A is mutually exclusive with the others. In the worst case, the greedy algorithm may select table A first, making it impossible to choose any of the other tables. This results in the table record plan with the smallest total entropy. If all remaining tables are independent, the upper bound for entropy is $(n-1) \cdot e$, which is $(n-1)$ times the entropy of the plan derived by the greedy algorithm. Thus, the greedy algorithm cannot guarantee optimal performance. A potential workaround involves running the greedy algorithm m times, each time starting with a different table from the array (from table I to table m), after sorting the tables in decreasing order of entropy. By comparing the entropy of these m plans, we can select the one with the highest entropy. However, in our evaluation, we observe that real programs typically have few mutually exclusive conditions. As a result, the greedy algorithm can produce good record plans for the P4 programs we tested.

V. Table Hit Situation

Previously we only considered the table execution information, this section takes the hit information into account.

In Section III, we noted that the table has three states, *not executed*, *executed but not hit* and *hit*. For the table at the *cut* node, it must be executed, so 1 bit is enough to record whether it is hit or not (e.g., A in Fig. 4); for other tables, II bits are required to record these three states. We assign 0, 1, or 2 bits to a table to record this information. When one bit is assigned to a *cut* table, this bit is used to record hit information; when one bit is assigned to a non-*cut* table, this bit is used to record execution information. On the other hand, when 2 bits are assigned to a non-*cut* table, the 2 bits can record all information. To find the optimal table record plan with both execution and hit information, we extend the algorithms introduced in previous sections.

A. How to Calculate Entropy

We extend Algorithm 1 to compute the entropy of both the execution and hit information. We still follow the framework of Algorithm 1, i.e., we first divide the DAG into subgraphs, calculate the entropy of each subgraph, and then sum them up. The difference is that we need to include the hit information entropy when calculating the entropy of the subgraphs.

In Algorithm 1, $path\_set$ saves each $path$ and its probability p, and we consider $p * log_2(p)$ as the information entropy of each $path$, but it only includes table execution information. Now, suppose the path consists of several tables containing only execution information and one table, denoted as $table_h$, which includes both execution and hit information. Let p_h represent the probability of $table_h$ being hit. The information entropy of the $path$ can then be expressed as

$$\begin{aligned}H_{path} &= pp_h log_2(pp_h) + p(1-p_h)log_2(p(1-p_h)) \\ &= pp_h log_2(p) + pp_h log_2(p_h) + p(1-p_h)log_2(p) \\ &\quad + p(1-p_h)log_2(1-p_h) \\ &= plog_2(p) + p(p_h log_2(p_h) + (1-p_h)log_2(1-p_h))\end{aligned}$$

We denote $H_{hit} = p_h log_2(p_h) + (1-p_h)log_2(1-p_h)$, then

$$H_{path} = plog_2(p) + pH_{hit}$$

We extend the number of tables with hit information to m, denote the tables as $table_{h_1}, table_{h_2}, \ldots, table_{h_m}$, the probability of $table_{h_i}$ being hit as p_{h_i}. Similarly, We denote $H_{hit_i} = p_{h_i} log_2(p_{h_i}) + (1-p_{h_i})log_2(1-p_{h_i})$. Then the information entropy of the $path$ can be generalized to

$$H_{path} = plog_2(p) + p\Sigma_{i=1}^{m} H_{hit_i} \tag{2}$$

pH_{hit_i} can be interpreted as the entropy of table hit information. We replace $p * log_2(p)$ in line 11 of Algorithm 1 with Equation 2, allowing the algorithm to compute the entropy with table hit information.

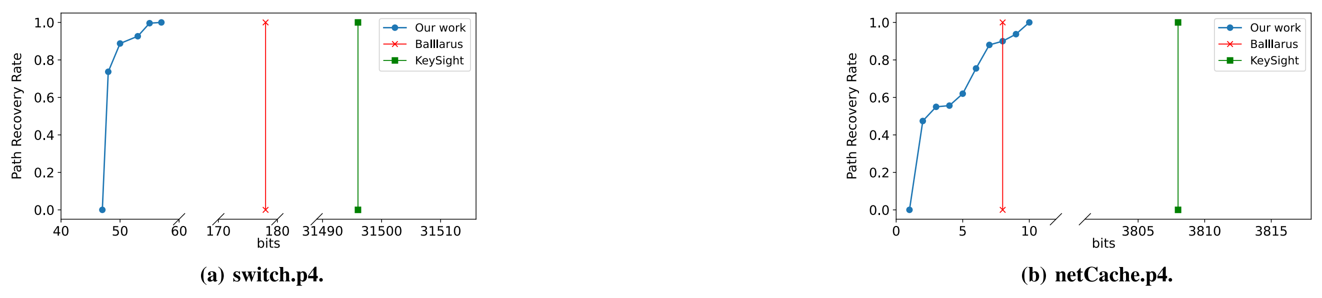
B. Extended Greedy Algorithm

When considering the table hit information, the search space expands too much, making it difficult for the search algorithm to solve the problem. Therefore, we extend the greedy algorithm in Section IV-C to find an *approximately optimal* solution. In the extended algorithm, we still choose the local optimal solution at each step, assigning a bit to the valid table that produces the most entropy. However, for different tables, the bit is used to collect different information.

For instance, there are three types of tables in Fig. 4. Table A and E are *cuts* tables, and the bit assigned to them will record hit information; tables B and C have already been assigned bits to record execution information, so the newly assigned bit will also be used to record hit information; table D is a non-*cut* table, and the bit assigned to it will record execution information. The algorithm finds the local optimal solution at each step until the available PHVs are exhausted or no valid assignment can be found.

VI. Evaluation

We conduct experiments on a commodity Barefoot Tofino switch [13] and evaluate our algorithm on various P4 programs for recording table information. We synthesize the transition probabilities of each edge and the hit probabilities of each table with random numbers. Section VI-A shows the relation between path recovery rate and the number of bits used. Section VI-B shows the resource overhead incurred by the record code. Section VI-C evaluates algorithms used to compute information entropy. Section VI-D shows the time overhead of these algorithms.



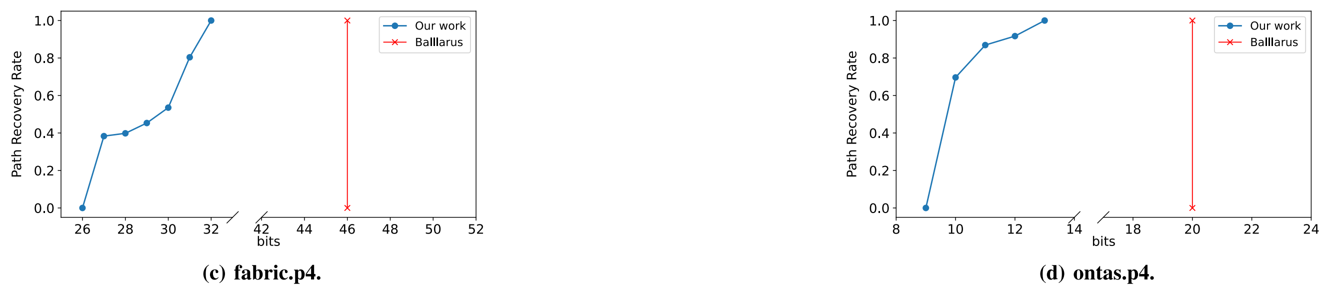


Fig. 8. Path recovery rate of different P4 programs.

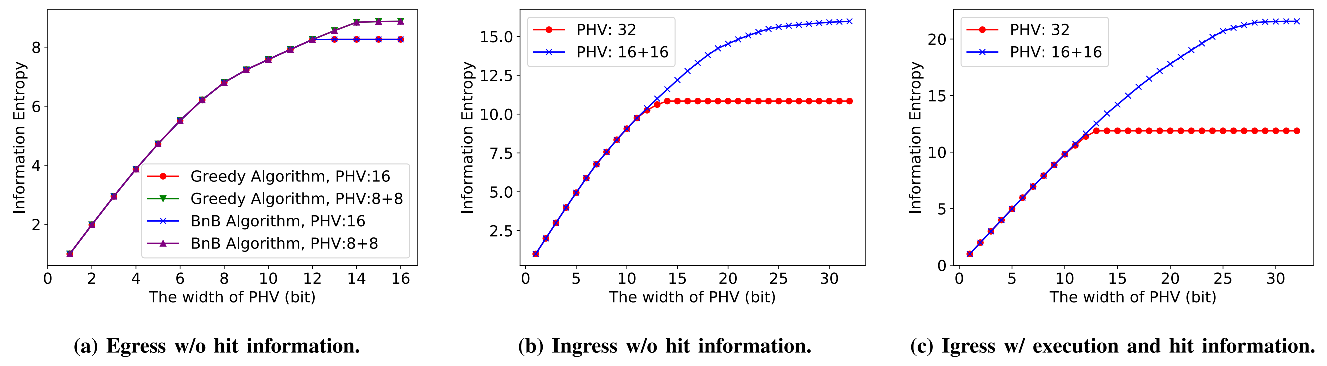


Fig. 9. Information entropy of different programs.

A. Path Recovery Rate

The definition of path recovery rate is as follows: given a p4 program with n possible execution path $path_1, \ldots, path_n$ with execution probability $p_1, \ldots, p_n$, our algorithm can identify k paths $path_{i_1}, \ldots, path_{i_k}$ among them. Then the path recovery rate is $\sum_{j=1}^{k} p_{i_j}$. We measure the relation between the number of used bits and the path recovery rate on the P4 program.

We evaluate our approach on several typical P4 programs and compare it to Balllarus [18] and KeySight [25]. Balllarus, which is most related to our work, employs Ball-Larus encoding to encode all possible paths. Thus, it consumes fixed-sized PHV bits given a P4 program. KeySight generates PostCards[1] [10] for each Packet Equivalence Class (PEC). Thus, we measure the mirrored packet size. We selected a large program, `switch.p4`, a medium-sized program, `fabric.p4`, and two small programs, `ontas.p4` and `netCache.p4`, for demonstration. The results are shown in the Fig. 8.

Compared to Ballarus, our approach reduces the number of bits required to achieve a full path recovery rate by 76%, 35%, and 30% for `switch.p4`, `ontas.p4`, and `fabric.p4`, respectively. This improvement is due to our entropy-guided method, which identifies certain paths that do not require encoding, whereas Balllarus encodes all possible paths. However, our approach uses 2 more bits than Balllarus for `netCache.p4`. This is because `netCache.p4` has fewer paths. In simple programs with fewer paths, the overhead of encoding all possible paths is smaller, and techniques like Ball-Larus coding can effectively minimize bit usage.

Compared to KeySight, our approach reduces bit usage by 99.86% and 99.73% for `switch.p4` and `netCache.p4`, respectively. This significant reduction is because KeySight relies on PostCards instead of directly embedding path information into the original packet. As a result, it has few restrictions on the size of PostCards and includes additional information within them, resulting in substantially higher bit consumption compared to our approach. Note that we do not show the results of KeySight in Fig. 8c and 8d. This is because the version of P4_16 supported by Keysight is relatively outdated. Unlike `fabric.p4` and `ontas.p4`, it does not support certain operations introduced in newer versions of P4_16, making it unable to analyze `fabric.p4` and `ontas.p4`. However, since Keysight generates specialized Postcards without the resource constraints associated with

[1]PostCards are mirrored packets that capture changes occurring at output ports, packet headers, and so on during packet processing by a switch.

TABLE II
RESOURCE OVERHEAD

p4 program	phv container(8 bits, 16 bits, 32 bits)	stage
netCache	8, 18, 14	10
netCache-path	10, 18, 14	10
ontas	20, 33, 27	7
ontas-path	23, 33, 27	9

TABLE III
SCALE OF SWITCH.P4

	All Table Num	Non-*Cut* Table Num	Dep Num
SwitchIngress	96	55	683
SwitchEgress	45	23	166

manipulating the original packets, it is expected to consume substantial resources, like the performance in `switch.p4` and `netCache.p4`.

More importantly, our work considers scenarios with constrained resources. Even under limited PHV resource conditions, our approach provides path execution information that exceeds the capabilities of other methods. For example, with just 48 bits, our algorithm achieves approximately a 75% path recovery rate on `switch.p4`, while Ballarus cannot record any paths with fewer than 178 bits, as it does not account for scenarios with limited available PHV resources.

B. Resource Overhead

We assess the impact of recorded code on resource overhead, primarily focusing on stage and PHV containers. We evaluate the *netCache* and *ontas* programs and refer to the programs inserted with the record code as *netCache-path* and *ontas-path*. For *netCache-path*, path information is recorded in metadata, while for *ontas-path*, it's recorded in the packet header.

Table II lists the resource overhead of the original programs and the programs inserted with the record code. The results show that *netCache-path* adds 2 PHV containers and *ontas-path* adds 3 compared to the original program. Moreover, regarding stages, *netCache-path* incurs no stage overhead, while *ontas-path* adds 2 stages. Our algorithm ensures that stage usage stays within hardware constraints (e.g., 12 in Tofino). The additional stage overhead arises from multiple tables writing to the same PHV container simultaneously, creating write-write dependencies and thereby increasing the number of required stages, as discussed in Section IV. Similarly, recording execution information in the packet header or metadata increases PHV container usage.

C. Information Entropy

Because `switch.p4` is a complex enough program containing numerous tables and dependencies, we utilize `switch.p4` to evaluate the two algorithms proposed for computing information entropy, i.e., BnB and Greedy. Table III shows the scale of the Ingress and Egress subprograms for `switch.p4`. The number of possible combinations in `switch.p4` is huge. For example, if we plan to assign 16 bits to SwitchIngress to record the execution information, then the number of possible combinations is significant, i.e., $C(55, 16)$, whose value is approximately $3 * 10^{14}$.

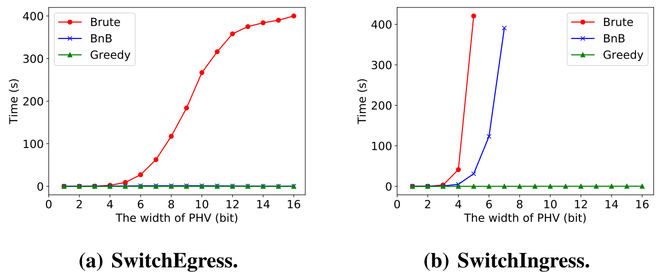


Fig. 10. Time overhead.

Fig. 9 illustrates information entropy with various PHV widths. PHVs are assigned to tables to record execution or hit information. "PHV: 32" indicates the use of a single 32-bit PHV container for recording, while "PHV: 16+16" implies the use of two 16-bit PHV containers. Fig. 9a depicts results from running the Greedy and BnB algorithms on **SwitchEgress** to identify maximum entropy for recording only execution information. The overlapping lines of the Greedy and BnB algorithms demonstrate the proposed Greedy algorithm's optimality on `switch.p4`. Similar results are obtained for other P4 programs, including `netCache.p4` and others.

Next, we explore **SwitchIngress** in scenarios with only execution information and with both hit and execution information. Fig. 9b and Fig. 9c present maximum entropy results achieved by recording only execution information and both execution and hit information, respectively, using the Greedy algorithm on the **SwitchIngress** subprogram. In both instances, employing two 16-bit PHV containers yields greater information entropy than a single 32-bit container. This aligns with our expectation, as multiple containers result in fewer write-write dependencies, facilitating the storage of more information.

D. Time Overhead

Finally, we examine the relation between the width of the used PHV containers and the time overhead imposed by different algorithms, including brute force (Brute), greedy (Greedy), and branch and bound (BnB). That is, we measure the time consumption of different schemes to generate a desirable table record plan while tuning the width of PHV. We evaluate different algorithms with the complex `switch.p4` and utilize two 8-bit containers for egress and a 16-bit container for ingress.

Fig. 10 illustrates the time overhead of different algorithms on `switch.p4`. We observe that Brute incurs the highest time overhead, followed by BnB and Greedy. This is because Brute enumerates all valid table sets to find the optimal table record plan, which is very time-consuming. With an increase in the number of used bits, Brute's time overhead is at least 8 times that of BnB, validating the efficacy of our pruning operation in BnB. Moreover, we observe that BnB's time overhead in the egress is nearly comparable to that of Greedy. However, in the ingress, BnB's overhead is noticeably higher than Greedy, due to the greater number of tables and the more complex program structure in the ingress pipeline, implying the efficiency and simplicity of Greedy.

VII. Related Work

P4-related troubleshooting. Recently, there has been P4DB [26], which is an on-the-fly debugging platform for programmable data plane that leverages additional debugging snippets to generate reports at runtime. It provides three primitives for network engineers: *Watch*, *Break*, and *Next*, thus providing three levels of visibility to help engineers debug bugs in P4 programs. The work in [27] showed that the performance query expressed in a domain-specific language could be translated to small pieces of code operating on different devices in the network collecting the necessary state. Most related to our work, Balllarus [18] proposed a method to track the execution of P4 programs in the data plane to help locate errors related to packet processing. It uses Ball-Larus codes to encode the paths of P4 programs for efficient path recording. However, it currently only considers tracking the table execution for all the possible paths, i.e., Balllarus can encode all N paths of a program in a single $\lceil log(N) \rceil$-bit variable. Hence, it cannot track certain tables within hardware constraints like our work. KeySight [25] proposes the Packet Equivalence Class(PEC) abstract and generates PostCards based on PEC. A PostCard can be viewed as a mirrored packet that encapsulates packet runtime information, meaning KeySight does not directly embed path information into the original packet. Additionally, the PostCard includes other runtime details, such as the output port, which increases its bit consumption. Recently, to overcome the limitations of current monitoring methods exhibit performance and granularity, [28] proposed P4-based implementation using In-band Network Telemetry, which is shown to exhibit minor networking and processing overhead. However, they attempt to record the full execution information of P4 programs, which takes too many PHV resources and is impractical for large-scale P4 programs.

Network troubleshooting system. Some existing network troubleshooting systems use mirroring to monitor the network. NetSight [10] mirrors each packet to a specific server and tags the packet with some metadata. EverFlow [12] adopts a "match and mirror" policy to mirror only a portion of the packet, thus reducing traffic overhead. NetSeer [11] utilizes the programmable data plane to detect and report flow-level events. Another work showed that network failures could be immediately detected and flushed out of the path information table at each server by a timeout mechanism [29]. NetSight and EverFlow do not consider programmable switches, and Net-Seer, while utilizing programmable switches to troubleshoot traditional network anomalies, still does not consider the new applications that programmable switches may undertake. We propose to append some internal table information to the mirrored packet to help troubleshoot.

Network telemetry. Some network telemetry systems [30], [31], [32] allow network engineers to express telemetry queries in a dataflow language, but these systems only operate on packets when they arrive at the switch. PacketScope [33] is the first work to provide visibility inside the switch. PacketScope is an extension of Sonata [31], and it can answer queries about internal processing, enabling network engineers to infer what is happening in a switch. However, PacketScope can only provide statistical information, not details about how each packet is processed in the switch. Causal telemetry [32] captures causal relationships between events, including those that occur on physically separated devices. Moreover, a P4-based prototype implementation is presented, which covers a case study that uses causal telemetry to detect Priority-Based Flow Control (PFC) deadlocks.

VIII. Conclusion

Effective troubleshooting requires recording real-time internal information of each packet in the programmable data plane. However, because of the hardware limitation, recording the table information arbitrarily is infeasible. Thus, we must select the informative tables carefully. To this end, we treat the P4 program as a probabilistic transition DAG and propose to use information entropy to evaluate a set of recorded tables. Then we take a two-step approach and design algorithms to find the table record plans. Experiments show that our algorithms can get the optimal and approximately optimal set of tables efficiently. As part of our future work, we will investigate variations of entropy metrics and explore how they align with the specific requirements of the programmable data plane. We aim to collaborate with industry partners to deploy the solution in operational networks and assess its effectiveness in production troubleshooting scenarios.

References

[1] H. Zheng et al., "FlyMon: Enabling on-the-fly task reconfiguration for network measurement," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 486–502.

[2] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Mar. 2022, pp. 743–759.

[3] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.

[4] A. Sapio et al., "Scaling distributed machine learning with in-network aggregation," in *Proc. USENIX NSDI*, 2021, pp. 785–808.

[5] X. Jin et al., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017, pp. 121–136.

[6] Z. Liu et al., "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. USENIX FAST*, Feb. 2019, pp. 143–157.

[7] T. Pan et al., "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proc. ACM SIGCOMM*, 2021, pp. 194–206.

[8] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, and H. H. Liu, "Cetus: Releasing P4 programmers from the chore of trial and error compiling," in *Proc. USENIX NSDI*, 2022, pp. 371–385.

[9] N. Zheng et al., "Meissa: Scalable network testing for programmable data planes," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 350–364.

[10] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, Apr. 2014, pp. 71–85.

[11] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 76–89.

[12] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, 2015, pp. 479–491.

[13] *Barefoot's Tofino*. Accessed: 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html

[14] X. Wu et al., "NetPilot: Automating datacenter network failure mitigation," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 419–430, Sep. 2012.

[15] F. Ruffy, T. Wang, and A. Sivaraman, “Gauntlet: Finding bugs in compilers for programmable packet processing,” in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Jan. 2020, pp. 683–699. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ruffy

[16] X. Chen et al., “Melody: Toward resource-efficient packet header vector encoding on programmable switches,” in *Proc. IEEE INFOCOM*, May 2023, pp. 1–10.

[17] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, “Modular switch programming under resource constraints,” in *Proc. USENIX NSDI*, 2022, pp. 193–207.

[18] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford, “Tracking P4 program execution in the data plane,” in *Proc. ACM SOSR*, 2020, pp. 117–122.

[19] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, 1948.

[20] P. Bosshart et al., “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[21] *P4 Software Switch*. Accessed: 2015. [Online]. Available: https://github.com/p4lang/behavioral-model

[22] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proc. Theory Pract. Softw., Int. Conf. Tools Algorithms Construct. Anal. Syst.* Cham, Switzerland: Springer, 2008, pp. 337–340.

[23] B. Tian et al., “Aquila: A practically usable verification system for production-scale programmable data planes,” in *Proc. ACM SIGCOMM*, 2021, pp. 17–32.

[24] J. Clausen, “Branch and bound algorithms-principles and examples,” Dept. Comput. Sci., Univ. Copenhagen, Copenhagen, Denmark, Tech. Rep., 1999, pp. 1–30.

[25] Y. Zhou et al., “KeySight: Troubleshooting programmable switches via scalable high-coverage behavior tracking,” in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 291–301.

[26] C. Zhang et al., “P4DB: On-the-fly debugging of the programmable data plane,” in *Proc. IEEE ICNP*, Oct. 2017, pp. 1–10.

[27] J. Khan and P. Athanas, “Query language for large-scale P4 network debugging,” in *Proc. ACM/IEEE ANCS*, Jul. 2018, pp. 162–164.

[28] H. N. Nguyen, B. Mathieu, M. Letourneau, G. Doyen, S. Tuffin, and E. M. de Oca, “A comprehensive P4-based monitoring framework for L4S leveraging in-band network telemetry,” in *Proc. NOMS-IEEE/IFIP Netw. Oper. Manage. Symp.*, May 2023, pp. 1–6.

[29] C. Jia et al., “Rapid detection and localization of gray failures in data centers via in-band network telemetry,” in *Proc. NOMS - IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2020, pp. 1–9.

[30] S. Narayana et al., “Language-directed hardware design for network performance monitoring,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 85–98.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: Query-driven streaming network telemetry,” in *Proc. ACM SIGCOMM*, 2018, pp. 357–371.

[32] Y. Liu, N. Foster, and F. B. Schneider, “Causal network telemetry,” in *Proc. 5th Int. Workshop P4 Eur.*, Dec. 2022, pp. 46–52.

[33] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, “PacketScope: Monitoring the packet lifecycle inside a switch,” in *Proc. ACM SOSR*, 2020, pp. 76–82.

Chengyuan Huang (Member, IEEE) received the B.Eng. and Ph.D. degrees from Beijing University of Posts and Telecommunications in 2015 and 2021, respectively. From 2021 to 2023, he was a Post-Doctoral Researcher with Purple Mountain Laboratories, Nanjing, China. He is currently an Assistant Researcher with the Department of Computer Science and Technology, Nanjing University. His research interests include data center networks, software-defined networking, and distributed systems.

Yibo Xiao received the B.S. degree from the Department of Computer Science and Technology, Nanjing University, China, in 2024, where he is currently pursuing the Ph.D. degree. His research interests include programmable networks and network system design.

Tianfan Zhang received the bachelor’s degree from Nanjing University of Aeronautics and Astronautics in 2020 and the master’s degree from the Department of Computer Science and Technology, Nanjing University, China, in 2023. His research interests include computer networks and distributed systems.

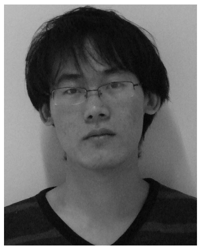
Chao Yang received the B.S. degree from the Department of Computer Science and Technology, Nanjing University, China, in 2022, where he is currently pursuing the M.E. degree. His research interests include in-network computing and congestion control.

Dong Zhang is currently the Chairperson of Jinan Inspur Data Company Ltd., has led the development of the world’s highest computing and storage density rack server, the first China UNIX operating system. He has made creative contributions in areas, such as converged architecture and high-end system software, earning one national award, and 11 provincial-level awards.

Bingheng Yan received the Ph.D. degree from Xi’an Jiaotong University in 2010. He is currently the Cloud Research and Development Director of Jinan Inspur Data Company Ltd., where he has led a wide range of virtualization research projects, and the development of Inspur’s server virtualization product InCloud Sphere, which break the global world record of SpecVirt. His research interests include operating systems, virtualization, and cloud computing.

Ahmed M. Abdelmoniem (Senior Member, IEEE) received the Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology (HKUST), Hong Kong, in 2017. He held the positions of a Research Scientist with KAUST, Saudi Arabia, and a Senior Researcher with the Huawei's Future Networks Laboratory (FNTL), Hong Kong. He is currently an Associate Professor with the School of Electronic Engineering and Computer Science, Queen Mary University of London, U.K., and leads the SAYED Systems Group. He is the Principal Investigator and the Co-Investigator on several national and international research projects, funded mainly by grants totaling over U.S. $1.5 million in funding. He has published numerous (more than 80) articles in top venues and journals in distributed systems, computer networking, and machine learning. His current research interests include optimizing systems supporting distributed machine learning, federated learning, and cloud/data-center networking, emphasizing performance, practicality, and scalability. He is a member of ACM and USENIX. He was awarded the prestigious Hong Kong Ph.D. Fellowship from RGC of Hong Kong in 2013 to pursue the Ph.D. degree at HKUST.

Gianni Antichi received the B.S., M.S., and Ph.D. degrees from the University of Pisa, Italy, in 2006, 2007, and 2011, respectively. From 2013 to 2018, he was a Post-Doctoral Researcher with the Department of Computer Science and Technology, University of Cambridge. He is currently an Associate Professor with the Politecnico di Milano, Italy, and the Queen Mary University of London, U.K. His research interests include data center networks, programmable hardware, end-host networking, network function virtualization, and distributed systems.



Xiaoliang Wang (Member, IEEE) received the Ph.D. degree from the Graduate School of Information Sciences, Tohoku University, Japan. From 2010 to 2014, he was an Assistant Professor with the Department of Computer Science and Technology, Nanjing University, China, where he is currently an Associate Professor. He has published more than 30 technical papers at premium international journals and conferences, including IEEE TRANSACTIONS ON INFORMATION THEORY, IEEE TRANSACTIONS ON COMMUNICATIONS, IEEE INFOCOM, USENIX ATC, and USENIX FAST. His research interests include network systems and optical switching networks.

Fu Xiao (Senior Member, IEEE) received the Ph.D. degree in computer science and technology from Nanjing University of Science and Technology, Nanjing, China, in 2007. He is currently a Professor and a Ph.D. Supervisor with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing. His research papers have been published in many prestigious conferences and journals, such as IEEE INFOCOM, IEEE ICC, IEEE IPCCC, IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, IEEE TRANSACTIONS ON MOBILE COMPUTING, ACM TECS, and IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY. His research interests include the Internet of Things and mobile computing. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Wanchun Dou received the Ph.D. degree in mechanical and electronic engineering from Nanjing University of Science and Technology, China, in 2001. From April 2005 to June 2005 and from November 2008 to February 2009, he visited the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, as a Visiting Scholar. He is currently a Full Professor with the State Key Laboratory for Novel Software Technology, Nanjing University. He has chaired three National Natural Science Foundation of China projects and published more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service computing.

Guihai Chen (Fellow, IEEE) received the B.S. degree in computer software from Nanjing University in 1984, the M.E. degree in computer applications from Southeast University in 1987, and the Ph.D. degree in computer science from The University of Hong Kong in 1997. He had been invited as a Visiting Professor with Kyushu Institute of Technology, Japan; The University of Queensland, Australia; and Wayne State University. He is currently a Distinguished Professor with Nanjing University. He has published more than 350 peer-reviewed articles and more than 200 of them are in well-archived international journals, such as IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, IEEE/ACM TRANSACTIONS ON NETWORKING, and *ACM Transactions on Sensor Networks*, and also in well-known conference proceedings, such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext, and AAAI. His research interests include parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering. He has won nine paper awards, including the ICNP 2015 Best Paper Award and the DASFAA 2017 Best Paper Award.

Hao Yin is currently a Changjiang Distinguished Professor with Tsinghua University. He has previously held roles as the Chief Scientist at China's largest content distribution network service provider, ChinaCache (NASDAQ: CCIH) and as the Deputy Director of the Ministry of Education-Microsoft Joint Key Laboratory. His research interests include computer networks, big data, and blockchain. He has been honored with awards, including the Second Prize of the National Technology Invention Award and the Second Prize of the National Natural Science Award.

Chen Tian (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He was an Associate Professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology. From 2012 to 2013, he was a Post-Doctoral Researcher with the Department of Computer Science, Yale University. He is currently a Professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include data center networks, network function virtualization, distributed systems, internet streaming, and urban computing.