Sirius: Composing Network Function Chains into P4-Capable Edge Gateways

Jiaqi Gao[†]*, Jiamin Cao*[†]*, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, Ennan Zhai*
Alibaba Cloud

Abstract

Alibaba Cloud designs and deploys P4-capable gateway to accelerate the processing of the diverse business traffics in the edge cloud. Since the programmable ASIC in the gateway only accepts a monolithic, pipelined P4 program, the dozens network function chains for different business traffics have to be composed into one. This is non-trivial due to the contention between the complexity of network function chains and the limited resource in the programmable ASIC. In this paper, we present Sirius, a system that automates network function chain composition process. Sirius synthesizes tables to identify which business traffic the input packet belongs to, pipelines loops in the merged network function graph via recirculations, and partitions the graph between programmable ASIC and CPU when the required memory consumption exceeds the ASIC's capability. So far, Sirius has automated network function arrangement in hundreds of gateways, and has effectively decreased our programmers' workload by three orders of magnitude, from weeks to minutes.

1 Introduction

As a cloud provider, Alibaba operates hundreds of edge clouds to deliver fast services (*e.g.*, game and video) to global end users. To maintain reasonable costs, each of these small-scale edge clouds contains a pair of gateways, a few switches, and tens of light-weight servers with tight space constraints and CPU compute limitations. Our typical edge cloud topology is shown in Figure 1(a). Depending on the services deployed, the edge cloud can serve more than 10 types of business traffics (load balancer, proxy, Virtual Private Cloud (VPC), *etc.*) Illustrated in Figure 1(b), each business traffic requires its own network function processing chain, different directions of the same business traffic may traverse network functions in different orders. As services today constantly evolve, it has become increasingly difficult for these resource-limited edge clouds to handle the ever-growing traffic and CPU overhead.

Recent advances in programmable switch ASICs have enabled us to offload network functions from edge cloud servers to the programmable switch ASICs. We designed and built our own P4-capable edge gateway. Shown in Figure 1(c), the gateway is equipped with a programmable switch ASIC and server-grade CPU with dozens of cores, connected via a pair of non-programmable NICs with hundreds of gigabits throughput. The gateway sits on the border of the edge cloud between the ISP and edge cloud switches. Deploying network function chains onto the P4-capable gateway significantly reduces the usage of constrained server CPU resources and improves the performance of our edge clouds.

Despite the P4-capable gateway's performance and flexibility, we face a tough programming challenge. Recent compiler works such as Lyra [5], Cetus [13] allow programmers to develop *one* network function chain fast and resource-optimized. However, programmers struggle with **composing all network function chains into the gateway while satisfying the throughput requirement.** More specifically, since the programmable ASIC is performant but limited in resources, the programmers have to merge all network function chains into a graph by reusing overlapping network functions, and maximize the network functions assigned to the programmable ASIC to process hundreds of gigabits of traffic flowing through the gateway. Network function chain composition involves three major challenges:

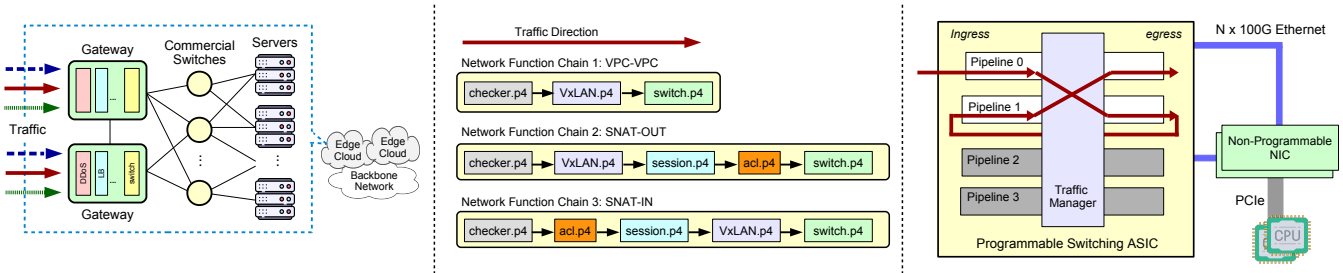
Identification. When a packet arrives, the first step is to identify which business traffic it belongs to and which network functions to execute. While there are many potential ways of identifying a packet, the solution should be generic and can be implemented with reasonable resource overhead.

Pipelining. In real-world applications, it is inevitable that different business traffics execute network functions in different orders. Given the ASIC's pipeline architecture, it is infeasible to compose these different processing chains directly. Such ordering conflicts can be resolved by inserting recirculations but with substantial human efforts.

Partitioning. Even after resolving ordering conflicts, the composed program containing all required network functions may not fit on the programmable ASIC due to its resource constraints. Thus, some components in the composed program have to be moved to the gateway's CPU, at the expense of lower performance and degraded traffic throughput. Such partitioning between programmable ASIC and the CPU is non-trivial because it requires careful balancing between the deployment capability and the overall throughput.

State of the art. To the best of our knowledge, no prior compiler work [3, 5–7, 9, 10, 12, 13, 19, 20, 23–25, 27] targeted the above-mentioned network function chain composition problem. μP4 [21] and P4 Weaver [4] enables modular P4 programming and composes multiple P4 modules into a directed acyclic graph (DAG) without reusing any common code block across modules. P4Visor [28], HyperV [26], Dejavu [25], and HyPer4 [8] merge multiple P4 programs into

[†] Both authors contributed equally to this paper



(a) Our edge cloud deployment topology (b) Three business traffics & network function chains (c) The architecture of the gateway in our edge cloud

Figure 1: Our edge cloud's topology, business traffic examples, and architecture of our P4-capable gateway.

one but do not handle ordering conflicts, or partition the program across ASIC and CPU. These approaches incur high memory overhead and cannot handle production-scale complexity with tens of business traffics and hundreds of match action tables.

Our system: Sirius. In this paper, we present Sirius, a system that automates network function chain composition in Alibaba's edge clouds. Sirius takes the following inputs: (1) P4 programs that define the network function chains for each business traffic, (2) a traffic identification database that describes flow characteristics for sets of business traffics, and (3) edge cloud topology and the throughput requirement for each traffic. Sirius then returns a composed P4 program that can handle all the above traffics and can deploy on the programmable ASIC. It also outputs the list of modules that must be moved to the CPU, so that our developers can later implement them as C++ programs.

Overall, this paper makes the following contributions:

- A synthesis algorithm that generates a memory-efficient P4 traffic identification table, along with corresponding guard conditions to identify each business traffic (§4).
- A pipelining algorithm that finds all possible candidates for resolving ordering conflicts in composing business traffic processing chains (§5).
- A new resource encoding paradigm and iterative searching approach to find the best pipelining and partitioning plan to minimize CPU load. (§6)

We have been using Sirius in production for one year, and it has automated the arrangement of network functions for hundreds of gateways in our global edge clouds. Sirius has effectively decreased our network function arrangement workload by three orders of magnitude (from weeks to minutes). In §7, we share our experience in using Sirius, representative cases solved by Sirius, and lessons we learned. We also evaluate Sirius's performance in §8.

2 Background

Edge clouds are deployed closer to end users and deliver dozens of services (*e.g.*, IoT, cloud gaming, CDN, and storage) with lower latency. Gateways on the edge cloud host network functions to process all services' business traffic at hundreds

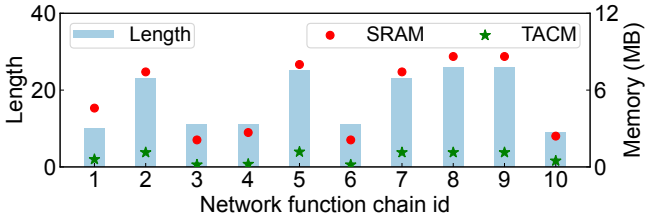


Figure 2: The length and resource consumption of overlapping tables of each network function chain.

of gigabits per second, such as balancing e-payment load, translating network address for VPC, collecting statistics for QoS and billing, *etc.* Traditionally, the network functions are implemented as software programs running on the CPU. As the scale of the edge cloud and the complexity of deployed services grow, the CPU-based solution struggles to keep up. We spent hundreds of CPU cores on an edge cloud site only processing the traffic traversing through, resulting in huge overhead both in cost and space. Therefore, we built our own P4-capable gateway and accelerated our network functions via the programmable ASIC. Depending on the services deployed, the gateway serves traffic at hundreds of gigabits to multi-terabits per second.

Developing atop the P4-capable gateway is challenging due to the scale and complexity of our network functions. To quantify the complexity, we examined four P4 programs for different edge cloud sites and summarized the line of codes and resource consumption in Table 1. We can see that all four programs occupy all 12 stages of the programmable ASIC, and all have high Packet Header Vector (PHV) usage. The largest program has 248 tables and consumes all four pipelines in the gateway's ASIC.

We experience such complexity and scale because each P4 program is composed of tens of network function chains processing different business traffic. We further dive into the 'Medium 1' program, which is composed of 10 network function chains. We examined the length of each network function chain and the result is recorded in Figure 2. The length varies from 9 to 26, which means some chain occupies at least 26 stages in the programmable ASIC. We also observe a huge amount of table overlapping between the chains. We recorded the SRAM and TCAM memory size of the tables shared with other chains and the result is shown in Figure 2. Each chain at least reuses 1MB of SRAM memory and 180KB

Scale	LoC	# of Table	# of Pipe.	PHV (%)	SRAM (%)	TCAM (%)	Stage
Small	4155	81	2	81.1%/68.8%	78.0%/98.8%	28.1%/10.1%	12/12
Medium 1	11870	197	2	88.9%/87.0%	51.7%/49.3%	64.6%/25.3%	12/12
Medium 2	9996	156	2	69.1%/85.4%	34.9%/47.2%	27.8%/29.2%	12/12
Large	16190	248	4	85.9%/97.3%/59.4%/87.4%	39.5%/67.2%/86.8%/66.9%	26.4%/33.0%/17.0%/35.4%	12/12/12/12

Table 1: The resource usages of four P4 programs at different scales.

of TCAM memory. If we choose not to reuse the tables and duplicate them, the overall memory consumption explodes by 3.56X in SRAM and 4.93X in TCAM, which way exceeds the ASIC's capability. Thus, it is necessary for us to compose network function processing chains that overlap with each other.

To the best of our knowledge, no prior work can handle the complex challenges we face. We believe the fundamental reason is that current abstractions either assume there exists only one huge network function chain (P4) or multiple chains are independent of each other (Lyra [5]). Such assumptions do not hold in our edge cloud scenario, as explained above.

Now we use the example network function chain shown in Figure 1(b) to illustrate the network function chain composition process and the challenges. The three network function chains create a simple edge cloud instance that allows bidirectional communication between private and public networks. *SNAT-IN* and *SNAT-OUT* chains process traffic going in and out of the edge cloud respectively, and *VPC-VPC* chain process traffic flowing between private networks. The expected throughput is 300 Gbps for all business traffics.

It is impossible to compose the three chains by directly merging their network functions into a single P4 program. As illustrated in Figure 3(a), each chain defines a mandatory ordering among its network functions. The orderings imposed by different chains conflict with each other, resulting in the two loops (highlighted in red) in the graph. In particular, *SNAT-IN* and *SNAT-OUT* traffic flow in reverse directions, thus, the two network function chains execute modules in reversed orders as well. Such conflicts are inconsistent with the pipelined architecture of the programmable ASIC. To successfully compose the three chains onto our P4-capable gateway, the following three challenges must be addressed.

Challenge 1: Pipelining network functions. Ordering conflicts commonly exist when composing diverse network function chains. We employ the recirculation feature to resolve these conflicts. Recirculation allows a packet to go through the programmable ASIC one more time, at the cost of reducing the overall processing throughput. For example, Figure 3(b) shows a pipelining plan that follows the order of the SNAT-OUT chain. This allows SNAT-OUT traffic to be processed in one pass, while introducing two recirculations for SNAT-IN traffic (*i.e.*, a packet visits checker and acl in round 0, session in recirculation round 1, and VxLAN and switch in recirculation round 2). As a result, SNAT-IN traffic's maximum throughput is reduced to 1/2 of the recirculation channel's bandwidth since every packet goes through the recirculation channel twice. Assuming that the recirculation channel has 400 Gbps bandwidth, this pipelining plan violates the throughput requirement (*i.e.*, 300 Gbps). Instead, a feasible pipelining plan is shown in Figure 3(c), where both SNAT-IN and SNAT-OUT traffics recirculate only once and guarantees 400 Gbps maximum bandwidth. Note that this is not the only feasible pipelining plan, swapping `VxLAN.p4` and `session.p4` in Figure 3(b) also satisfies the throughput requirement.

Challenge 2: Identifying business traffics. Despite the fact that all network functions are present in the composed programmable ASIC, different business traffics visit different network functions defined by their processing chains. This requires us to identify which business traffic an input packet belongs to and only execute relevant P4 modules. Figure 3(d) shows a naive solution that inserts different identification tables before each module. For example, the two identification tables marked in red ignore *VPC-VPC* traffic according to the network function chain. The rest of the identification tables marked in blue allow all traffic to go through but also ensure that the table only executes at the correct recirculation round. However, this solution incurs high memory overhead and may lead to a longer execution chain of P4 tables. According to findings in Cetus [13], this approach does not scale.

Challenge 3: Partitioning network functions. In many cases, the composed P4 program simply requires too much memory resource and does not fit on the programmable ASIC. We have to partition the program and move some tables to the CPU. This requires careful balancing between resource consumption on the programmable ASIC and the throughput degradation caused by moving to the CPU. Furthermore, since the CPU sits on the recirculation path, a careless partitioning plan can introduce additional recirculations. For example, if we assign `switch.p4` to the CPU based on the pipelining plan in Figure 3(c), all three business traffics have to recirculate once more time to be forwarded out of the gateway. A better plan is to assign `session.p4` to the CPU since it already sits on the recirculation path, which we will detail in §6.1.

The network function composition problem causes significant overhead during gateway development. It often takes weeks for our programmers to find an arrangement plan, which is subject to change when the network function implementation or the gateway configuration (*e.g.*, set of supported business traffics) changes. Thus, it is necessary to build a system to automate the network function composition process.

3 Sirius Overview

Figure 4 presents Sirius's architecture. Sirius offers our programmers a set of high-level interfaces to automatically address the network function composition problem. The inter-

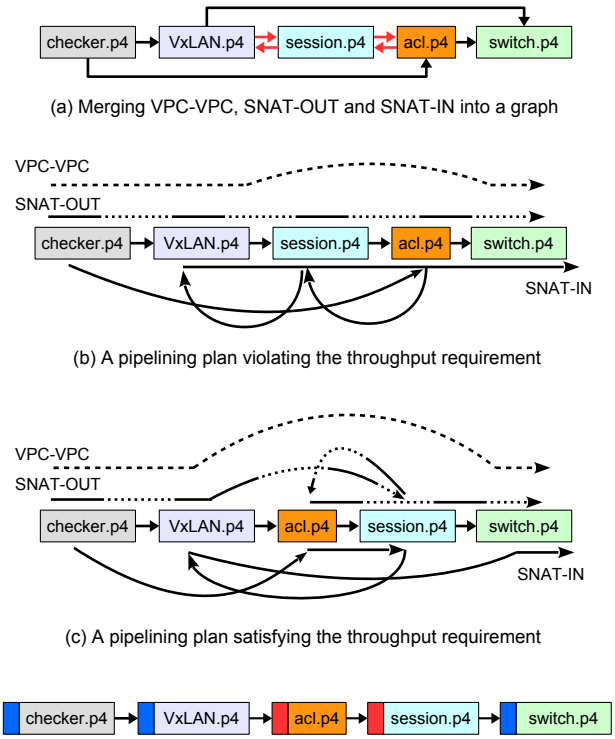


Figure 3: An example of a merged graph and two pipelining plans for this graph.

faces require the following input: (1) P4 programs that specify all the individual network functions and how these network functions are chained for diverse business traffics, (2) a traffic identification database, which contains flow characteristics rules (flow predicates) that distinguish sets of business traffics, (3) edge cloud topology and throughput requirements, which specify the throughput needs of different business traffics.

As shown in Figure 5, given a set of original P4 code for each network function chain (Figure 5(a)), Sirius follows three phases to produce the final composed P4 program that can compile and deploy to a programmable ASIC.

In the first phase, Sirius leverages the input traffic identification database to generate a *traffic identification table* that tags each packet according to a selected set of flow predicates. In addition, it generates guard conditions that utilize these tags to distinguish each business traffic (Figure 5(b)), such that a network function is only visited by traffic chains it belongs to. To accommodate memory constraints on the programmable ASIC, we describe in §4 our algorithm for synthesizing a memory-efficient traffic identification table.

In the second phase, Sirius generates solutions for resolving network function ordering conflicts among different processing chains. This is a unique challenge for composing network function chains onto a programmable ASIC, which follows a pipelined architecture. Sirius resolves such conflicts by inserting recirculations after certain network functions. To find

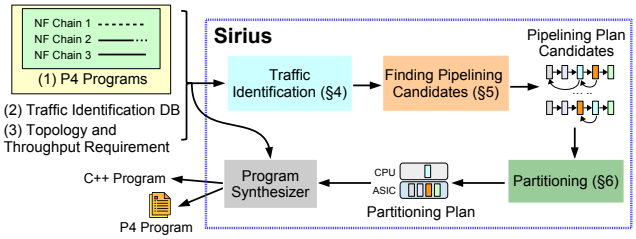


Figure 4: Sirius's system architecture overview.

pipelining plan candidates (*i.e.*, candidate recirculation points) with a satisfied number of recirculations *efficiently*, Sirius models this pipelining process as a feedback arc set problem and introduces an algorithm as explained in §5. Figure 5(c) illustrates the deployment of a candidate solution, where the guard condition before network functions are extended with predicates on the recirculation count.

In the third phase, if no pipelining candidate can fit within the switch resource, Sirius searches for P4 tables to assign to the CPU. We propose a novel *logical stage* encoding paradigm to transform this search into a satisfiability problem, and employ an iterative strategy to generate the optimal partitioning between the programmable ASIC and the CPU (§6). As illustrated in Figure 5(d), a solution may involve moving the session module to the CPU and adjusting recirculation destinations accordingly.

In this way, Sirius generates a composed P4 program that accommodates all original network function processing chains and can compile to the programmable ASIC.

4 Traffic Identification

The composition of multiple traffic processing chains requires the integrated program to distinguish business traffics from each other and add corresponding guard conditions at the entry of each module. This ensures a packet only visits modules that belong to its processing chain. Because the programmable switch has limited resources, it is necessary to reduce the memory usage overhead of traffic identification and to fit as much business traffic processing logic as possible.

In this section, we first show the input to Sirius, *i.e.*, the traffic identification database that records how header fields and metadata identify different business traffics (§4.1). Then, we introduce the insight that Sirius uses to synthesize the traffic identification table and add guard conditions on the programmable switch (§4.2).

4.1 Traffic Identification Database

Each business traffic has its unique flow characteristics. For example, VPC-VPC traffic travels within the internal network, and thus its source and destination IP are both in the internal IP range, while for SNAT-IN traffic, its source IP belongs to

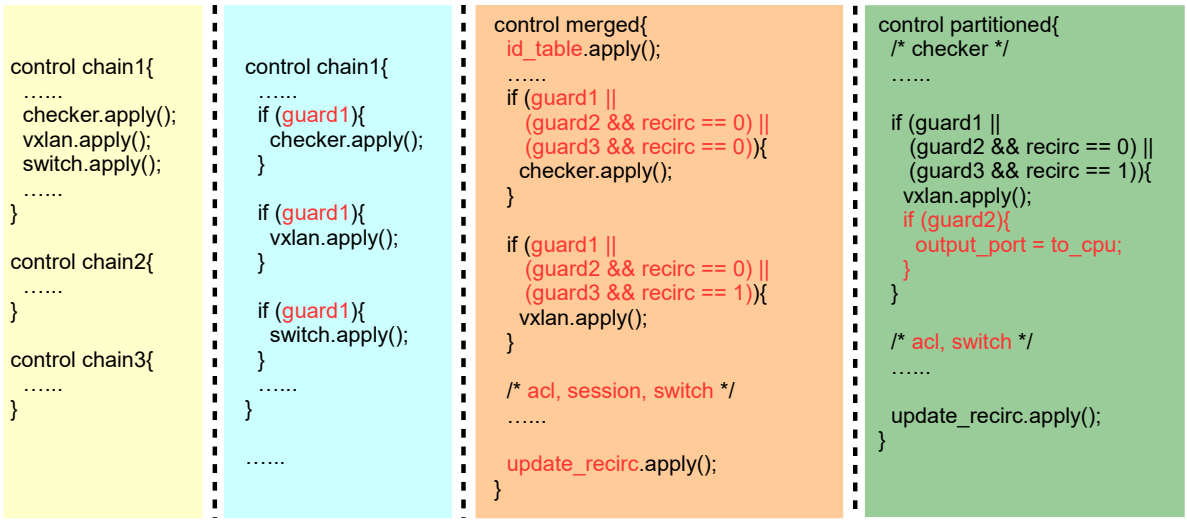


Figure 5: Changes on the P4 implementation after each phase. (a) depicts the original P4 code for each network function processing chain. (b) depicts chain 1 (VPC-VPC) after going through the traffic identification phase, where guard1 denotes the synthesized guard condition for it. (c) depicts the merged P4 code after going through the pipelining phase, assuming the pipeling solution shown in Figure 2(d). Here, recirc is a variable recording the number of recirculations experienced by a packet. (d) depicts the P4 code after going through the partitioning phase. Here, traffics belonging to chain 2 will be directed to the CPU after the vxlan module, which implements the solution in Figure 7(b).

Rule ID	Flow Predicate	Business Traffics	Entries
Rule 1	src_ip ∈ Public IP	SNAT-IN	100
Rule 2	src_ip ∈ VPC IP	VPC-VPC ∪ SNAT-OUT	10
Rule 3	dst_ip ∈ Public IP	SNAT-OUT	100
Rule 4	dst_ip ∈ VPC IP	VPC-VPC ∪ SNAT-IN	10
Rule 5	phy_port ∈ Internal Port	VPC-VPC ∪ SNAT-OUT	48
Rule 6	(src_ip, dst_ip) ∈ VPC IP PAIR	VPC-VPC	50

Table 2: Our example's traffic identification DB.

the public IP range. Sirius relies on the traffic identification database to maintain this information.

Sirius's traffic identification database adopts a flow-predicate-centric approach, *i.e.*, the primary key of the database is the membership relation of header fields (such as the source IP field) or metadata (such as the physical port ID on the switch) that distinguishes a subset of the business traffics (being a necessary and sufficient condition). An example traffic identification database is shown in Table 2. Rule 2 means if a traffic's source IP belongs to the VPC IP set, then it is either VPC-VPC or SNAT-OUT traffic, and vice versa. Besides the basic flow predicate and the corresponding business traffic set, the database also provides the number of entries in each rule. Three factors together decide the memory consumption of each rule. For example, Rule 6 occupies 10× more memory than Rule 2 in Table 2 because Rule 6 checks both source and destination IP and has 5× more entries[1].

We rely on the composition of flow predicates to distinguish each business traffic. For example, (Rule 2 AND Rule 4) separates VPC-VPC traffic from the rest, and (Rule 4 AND (NOT Rule 1)) also achieves the same goal. The reason for such a design is twofold. On the one hand, it is not always feasible in practice to find a single predicate to identify each business traffic. On the other hand, although Rule 6 uniquely defines VPC-VPC traffic, it requires more memory resources than composing Rules 2 and 4.

Implementation: parallel matching on all predicates We use parallel matching to implement the above flow predicate composition in a generic way. As illustrated in Figure 6, when a packet arrives at the programmable switch, we match it against all flow predicates in parallel and store the results in corresponding flags. Upon subsequent network functions, guard conditions are generated by composing these predicates, *e.g.*, `hdr.p2 && hdr.p4` denotes (Rule 2 AND Rule 4), which uniquely identifies VPC-VPC traffic.

4.2 Synthesis of Traffic Identification Tables

As discussed above, we draw flow predicates from a developer-provided traffic identification database (*e.g.*, Table 2) and use their logical compositions to identify each business traffic. This section describes how we choose the set of flow predicates (*i.e.*, rules) that is both *complete* (able to distinguish each business traffic) and memory efficient when implemented on the programmable switch[2].

[1] Rule 6's size is smaller than 90 (*i.e.*, 10 × (10 - 1)) because some source and destination IP pairs are not possible in reality.

[2] We ignore other types of resources (especially PHV) because (1) according to our experience, memory is the major bottleneck, and (2) it is challenging to quantify other resources' overhead and hard to choose one solution over another.

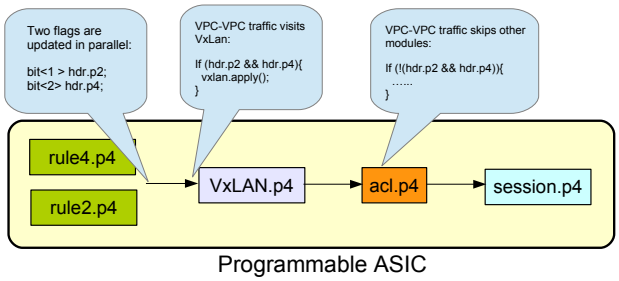


Figure 6: Implementation of flow predicates: rule 2 and rule 4

Our synthesis algorithm takes the traffic identification database as input and returns a set of rules as output that is complete and incurs minimum resource usage. The pseudo-code is listed in Algorithm 1. First, the algorithm iterates through each rule and builds a search tree from all the rules behind, where each node in the tree represents a subset of the rules in the database, and each branch adds one more rule to the subset (Line 3-4). There are $|\mathcal{R}|$ trees in total. Next, we iterate through all nodes in the search tree and find a node that represents the minimum valid set of rules (Line 5-15).

Because the resource consumption increases monotonically as rules are added, we apply three more optimizations to prune the search tree. Before searching, we sort the rules in the database by memory size and traverse them in ascending order (Line 2). Within a tree, we sort the nodes in the tree by their resource consumption in ascending order (Line 4) and stop searching when we find the first valid set (Line 19-22). Between trees, we stop searching the current tree when the current set has higher resource consumption than the global best solution (Line 16-18).

Example. Assume the business traffics shown in Figure 1 and the traffic identification database shown in Table 2. When searching directly without the optimizations, we need to check the search tree of each rule. For example, in the first round, we check the search tree of $\{Rule\ 1\}$ and obtain three traffic identification tables: $\{Rule\ 1, Rule\ 3\}$ with $(100\times32+100\times32)$ bits, *i.e.*, 800 bytes of memory usage, $\{Rule\ 1, Rule\ 4\}$ with $(100\times32+10\times32)$ bits, *i.e.*, 440 bytes of memory usage, and $\{Rule\ 1, Rule\ 6\}$ with $(100\times32+50\times64)$ bits, *i.e.*, 800 bytes of memory usage. After we check all search trees in all rounds, we choose $\{Rule\ 2, Rule\ 4\}$ with 80 bytes of memory usage generated in the second round. With the optimizations, we directly check the search tree of $\{Rule\ 2\}$ and obtain $\{Rule\ 2, Rule\ 4\}$, since $\{Rule\ 2\}$ has the smallest memory usage and $\{Rule\ 2, Rule\ 4\}$ has the smallest resource consumption in the tree.

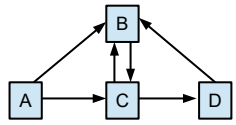
5 Finding Pipelining Candidates

A unique challenge in composing multiple network function processing chains is the inevitable ordering conflicts between individual network functions. Manifested as loops in the merged network function graph, these ordering conflicts commonly exist because business traffic going in and out of the gateway usually visits network functions in opposite directions. A network function graph with loops cannot be deployed on the programmable ASIC directly due to the ASIC's pipelined architecture. Sirius solves this problem by inserting recirculations in the merged network function graph. In this section, we first explain that the problem resembles a feedback arc set problem (§5.1), and next, we show how Sirius solves the problem and finds all pipelining candidates (§5.2).

Algorithm 1: Traffic identification table synthesis

Input: $\mathcal{T}$: Business traffic.
Input: $\mathcal{R}$: Rules in traffic separation DB.
Output: $\mathcal{F}$: Rules to synthesize traffic identification tables.

1 $\mathcal{F} \leftarrow \mathcal{R}$
2 $\mathcal{R} \leftarrow SizeSortAscending(\mathcal{R})$
3 **foreach** $r_i \in \mathcal{R}$ **do**
4 $\mathcal{C} \leftarrow SizeSortAscending(Combinations(\{r_i, r_{i+1}, \cdots, r_{|\mathcal{R}|}\}))$
5 **foreach** $\mathcal{R}' \in \mathcal{C}$ **do**
6 $\mathcal{X} \leftarrow \{\{t, t \in \mathcal{T}\}\}$
7 $\mathcal{S} \leftarrow \emptyset$
8 $terminate \leftarrow False$
9 **foreach** $r_k \in \mathcal{R}'$ **do**
10 **foreach** $x_j \in \mathcal{X}$ **do**
11 $tmp_1 \leftarrow x_j \cap r_k.action$
12 $tmp_2 \leftarrow x_j - tmp_1$
13 **if** $((tmp_1 \cup tmp_2) - \mathcal{X}) \neq \emptyset$ **then**
14 $\mathcal{X} \leftarrow tmp_1 \cup tmp_2$
15 $\mathcal{S}.append(r_k)$
16 **if** $RuleSize(\mathcal{S}) \geq RuleSize(\mathcal{F})$ **then**
17 $terminate \leftarrow True$
18 **break**
19 **if** $AllElementSizeOne(\mathcal{X})$ **then**
20 $\mathcal{F} \leftarrow \mathcal{S}$
21 $terminate \leftarrow True$
22 **break**
23 **if** $terminate$ **then break**
24 **return** $\mathcal{F}$



(a) An example merged network function graph

// All existing processing loops
L = { [B, C, B], [B, C, D, B] }
// Key-Value Table V

Execution Sequence	Loops
B → C	{ [B,C,B] , [B,C,D,B] }
C → B	{ [B,C,B] }
D → B	{ [B,C,D,B] }
A → C	{ }
… …	… …

(b) Vector and Key-Value Table

Figure 7: An example for our insight and modeling.

5.1 Modeling Pipelining Process

It is impossible to deploy a set of network functions containing ordering conflicts (*i.e.*, loops in the merged graph) on

the programmable ASIC directly. We address this issue by inserting recirculations at the end of certain network functions, which effectively breaks the immediate ordering (*i.e.*, removes the edges) after these network functions. Until there is no longer ordering conflict (*i.e.*, the merged graph becomes acyclic), this set of network functions becomes deployable on the programmable ASIC. Figure 5(c) illustrates that multiple processing chains can be merged into a single P4 program after resolving all ordering conflicts.

We use Figure 7(a) to explain this process. In Figure 7(a), there are four network functions *A*-*D*, and each directed edge represents an immediate ordering between two network functions, such as $B \rightarrow C$, required by the network function processing chain it belongs to. There are two processing loops, $B \rightleftarrows C$ and $B \rightarrow C \rightarrow D \rightarrow B$. If we remove the ordering $B \rightarrow C$, the two processing loops disappear. Removing $D \rightarrow B$ only removes one processing loop $B \rightarrow C \rightarrow D \rightarrow B$.

This tells us that we can leverage the recirculation feature (a commonly used feature in P4 for processing a packet multiple times) to "remove" a network function ordering, which eventually removes processing loops. At the same time, we should use recirculations as little as possible to avoid throughput drop. Therefore, our goal is to remove all processing loops with minimal recirculations, making the network function graph a directed acyclic graph.

This resembles the feedback arc set problem [2], which removes edges from a directed graph until it becomes acyclic. However, the main difference is that the original problem formulation focuses on minimizing the number of edges removed or by minimizing a certain weight. As a comparison, we ultimately focus on the traffic throughput, which is determined by a lot more factors, including the number of edges removed on each processing chain (*i.e.*, the number of recirculations).

5.2 Generating Pipelining Candidates

There are multiple pipelining candidates (*i.e.*, set of edges to remove) for the same merged graph. For example, both $\{B \rightarrow C\}$ and $\{C \rightarrow B, D \rightarrow B\}$ are valid solutions for Figure 7(a). However, it is hard to tell which one is better without going through the partitioning phase. In order to reduce the number of candidates entering the partition phase, Sirius employs the following two pruning strategies during the search. (1) The search backtracks if the current solution violates the **throughput** requirement, since inserting more recirculations would only result in lower throughput. (2) The search backtracks when a **valid** solution is found, such that unnecessary recirculations are prevented.

To generate the above sets of candidates, Sirius first computes a set L, which contains all processing loops, and a key-value table V. Figure 7(b) illustrates an example for L and V. In V, each key (say k_i) represents an execution sequence, and the corresponding value records all the processing loops that disappear once k_i is removed. It then searches for all sets of edges that collectively remove all loops in the graph, *i.e.*, for any set of edges $\mathcal{K}$, $\bigcup_{k \in \mathcal{K}} V[k]$ must equals L.

Sirius introduces a dynamic programming approach to explore the search space and applies heuristics based on the throughput specifications. The algorithm works as follows:

- (1) Remove the keys with an empty value from the key-value table V, and sort the keys based on the number of elements in the mapped values in descending order. For example, execution sequence $\{B \rightarrow C\}$ has two elements.
- (2) In the ranked V, traverse each key k_i and add it to $\mathcal{K}$, which denotes that the edge k_i is removed.
  - (2.1) Then (still for k_i), iterate each network function chain and compute the expected throughput given the current $\mathcal{K}$. When a chain contains n keys in $\mathcal{K}$, the chain's throughput is reduced to T/n, where T is the recirculation channel's throughput. If the expected throughput is violated, remove key k_i from $\mathcal{K}$, and jump to (3).
- (3) Select key k_{i+1} from the remaining elements in the ranked V, continue step (2) until the values of $\mathcal{K}$ covers L, then $\mathcal{K}$ is a solution (*i.e.*, a pipelining candidate). Continue the iteration until all solutions are found.

6 Partitioning

Given a set of pipelining plan candidates that have resolved all ordering conflicts and also satisfied the throughput requirements considering all inserted recirculations (generated from §5), we now need to compile and fit them into the programmable ASIC. Based on our one-year experience, none of the merged P4 code (*e.g.*, Figure 5(c)) can directly compile to the programmable ASIC due to its limited resources. Sirius thus proposes a partitioning approach that tries to move some tables or an entire network function to the CPU so that the resulting P4 code can be successfully compiled.

Similar to Lyra [5] and Cetus [13], our partitioning approach leverages the SMT solver to search for a satisfied result (*i.e.*, tables or network functions to move to the CPU); however, our scenario needs to take into account many different factors such as the recirculation feature and the CPU. Existing SMT solver-based approaches do not address these unique challenges.

In this section, we present a novel logical stage encoding approach (§6.1) that transforms partitioning problems into SMT problems. Then, we explain how we use iterative searching to find the best partitioning plan that minimizes the load on the CPU (§6.2).

6.1 Partitioning Encoding

Moving some tables or the entire network function to the CPU may add additional recirculation to the input network function chain, because a packet needs to first leave the programmable ASIC, then enter the CPU, and finally enter the programmable

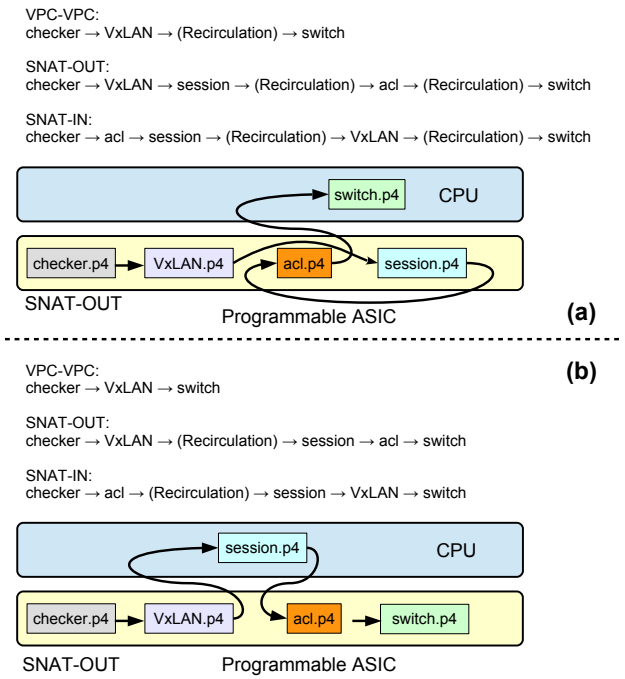


Figure 8: Recirculation situations resulting from two different partitioning plans.

ASIC again in a recirculation. If a partitioning plan is not well-chosen, the entire network function chain may include additional recirculations (introduced by the CPU), violating the throughput requirement. Figure 8 shows an example.

Suppose a programmable ASIC can only hold four network functions in Figure 3(a), and we have a pipelining result shown in Figure 3(c). Because the entire network function chain shown in Figure 3(c) cannot be put in the programmable ASIC, we need to move a network function to the CPU. Figure 8(a) shows a partitioning result that moves the `switch.p4` to the CPU. In such a partitioning plan, unfortunately, VPC-VPC, SNAT-OUT, and SNAT-IN have one, two, and two recirculations, respectively. Namely, this partitioning additionally adds a recirculation to each of these three chains. As a result, the throughput is significantly decreased. Figure 8(b) shows a better partitioning plan that moves `session.p4` to the CPU. In this case, all three network function chains achieve the desired throughput without any additional recirculation caused by the CPU.

Encoding. It is common to address resource allocation problems by encoding relevant constraints as math formulas and invoking SMT solvers to find a satisfying solution. However, encoding the effect of moving P4 tables to the CPU is not straightforward due to its intertwining with existing recirculations generated in §5.

We propose a concept called *logical stage* to address this challenge. While a physical stage describes where a *P4 table* is actually deployed, the logical stage depicts its execution sequence, *i.e.*, physical stage + N × recirculation rounds, where N denotes the number of physical stages in the switch.

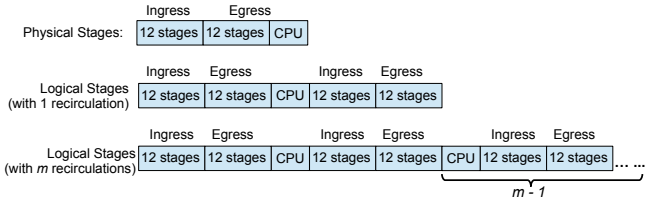


Figure 9: Physical stages and their logical stages.

As illustrated in Figure 9, Sirius duplicates m ingress and egress pipelines to model the logical execution sequence that spans m recirculation rounds. Further, Sirius inserts a new CPU stage between each egress and ingress pipeline to model the execution on the CPU.

The partitioning phase takes the merged P4 code (*e.g.*, Figure 5(c)) as input and calculates the set of tables to move to the CPU that enables the successful deployment of the remaining P4 tables. We encode this phase as a satisfiability problem as below.

Input: We use k to denote the total number of stages on the target ASIC. For the particular pipelining candidate, we use r_c to denote the number of recirculations for chain c.

Output: For each table t, s_{pt} and s_{ct} represents its physical and logical stage index, respectively.

Constraints:

- ASIC stage constraint: a table can either deploy at a physical stage or be moved to the CPU (the $(k+1)^{\text{th}}$ stage). Thus, $0 \le s_{pt} \le k$ must hold.
- A physical stage must correspond to a logical stage at a particular recirculation round: $(s_{ct} < k+1 \implies s_{pt} = s_{ct}) \wedge (k+1 \le s_{ct} < 2k+2 \implies s_{pt} = s_{ct} - k - 1) \wedge \cdots$ must hold.
- Moving tables to the CPU does not result in more recirculations: for each table t belonging to chain c, $0 \le s_{ct} \le (r_c+1) \times (k+1)$ must hold.
- Ordering constraint: if a chain orders table t after t', $s_{ct} > s_{ct'}$ must hold.

In addition to the above partitioning constraints, we also apply resource constraints defined in Lyra [5] and Cetus [13] to determine whether the remaining tables can indeed deploy to the programmable ASIC.

6.2 Finding the Partitioning Result

Given the above encoding, we invoke an SMT solver to search for a partitioning result with the optimization goal of **minimizing the CPU usage**. The performance of a program running on the CPU is affected by many factors and is hard to predict. Recent works such as Bolt [11] require packet traces to predict performance at reasonable accuracy. Instead, Sirius uses two heuristic metrics to achieve the goal: (1) minimizing the number of tables assigned to the CPU[3], and (2) when two

[3]We choose not to assign TCAM tables to the CPU since ternary matching's complexity is $O(n^{\log_3 2})$ [1] while LPM and exact match's complexity is only O(1) [18]

plans assign the same number of tables, choose the one with lower traffic volume.

In particular, Sirius adopts iterative searching and increases the allowed number of tables to move to the CPU gradually until a feasible plan is found by the SMT solver.

Iterative searching. The iterative searching reduces an optimization problem into a series of satisfiability problems, which the SMT solver is good at. Atop the existing encoding, for each table t, Sirius introduces one 0-1 variable i_t that denotes whether the table should be deployed on the CPU, which must satisfy: $(s_{pt} = k+1 \implies i_t = 1) \wedge (s_{pt} < k+1 \implies i_t = 0)$. Then the summation of all such variables $I = \sum_t i_t$ means the total number of tables assigned to the CPU. Then, Sirius limits the summation $I == m$ starting from $m == 0$, and checks whether there exists a partitioning plan $Part_m$. If not, this means no solution can be found when assigning m tables to the CPU, and then Sirius increases m by one and calls the solver again. If $Part_m$ exists, Sirius records it as a candidate. In case multiple solutions exist, we remove $Part_m$ from the solution set by negating it atop the encoded formula E, and check whether $E \;\&\; \neg Part_m$ is solvable. We continue the above process until all candidates under m are found. Among all the pipelining candidates that assign m tables to the CPU, Sirius selects the result with minimal CPU usage.

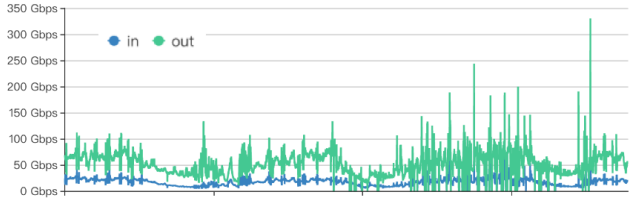
7 Experience

This section shares the deployment experience of Sirius (§7.1), real cases with Sirius (§7.2), and our lessons (§7.3).

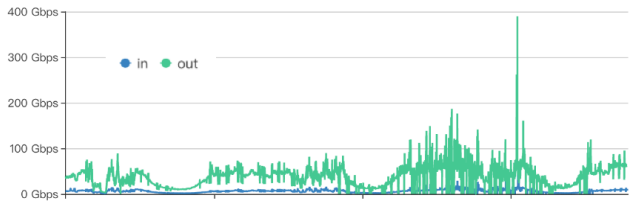
7.1 Deployment Experience

We started to deploy the edge clouds in 2017. As the number of business traffics and offloaded functions grew, it became increasingly difficult to arrange network functions in our edge clouds manually. We, therefore, started building Sirius in 2020. So far, Sirius has been used for one year. The network functions arranged by Sirius have been used to carry $O(10)$ types of business traffics, including streaming, games, IoT devices, and e-payments. With Sirius's assistance, we have built $O(100)$ gateways for edge-cloud services, $O(100)$ CDN nodes, and nearly 100 nodes for security in the past year. The peak throughput was higher than 10 Tbps across all edge-cloud instances.

Performance of gateways arranged by Sirius. An important metric for evaluating the effectiveness of Sirius is throughput, namely, whether the throughput of network functions arranged by Sirius meets our expectations. Figure 10 randomly selected two gateways where we used Sirius to arrange network functions automatically. These two gateways have been deployed in two edge clouds, respectively, and are mainly used to carry the streaming service. The throughput requirements for all the traffics on these two gateways are 500 Gbps. Figure 10 shows the performance of the gateways within one week. We



(a) The performance of gateway in the edge cloud A.



(b) The performance of gateway in the edge cloud B.

Figure 10: One-week performance of two edge clouds (for streaming service) where the network functions are arranged by Sirius. In - Traffic going into the edge cloud. Out - Traffic going out of the edge cloud.

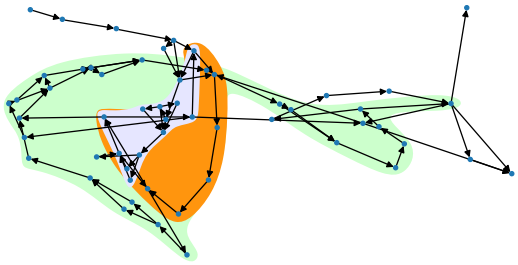
can observe that the peak traffic within one week was 300-400 Gbps, and the network functions arranged by Sirius can handle the traffic with very stable performance.

Development workload saved by Sirius. In terms of development efficiency, we combined Sirius with Lyra [5] to directly generate the compilable P4 programs that meet our specified throughput requirement. For the CPU-side code, our programmers have internal scripts to automatically generate C++ code. Before using Sirius, our gateway engineering team (more than twenty persons) spent more than two weeks analyzing and discussing a network function arrangement plan. After the initial network function arrangement version, it further took more than two weeks for adaptation and resource optimization. On the contrary, using Sirius, our programmers only need to write simple, high-level P4 programs and specify the throughput requirement. Sirius generates the expected results in an efficient and automatic way (within a few minutes). In the past year, Sirius decreased our programmers' workload by three orders of magnitude (from weeks to minutes).

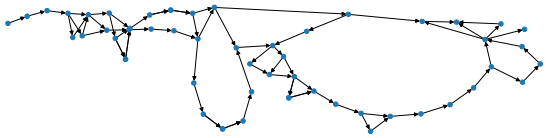
7.2 Real Cases Addressed by Sirius

We now describe three representative cases to show the practicality of Sirius in real world.

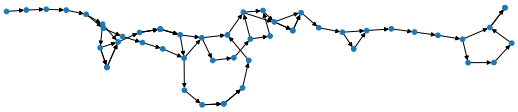
Merging and arranging network function chains for $O(10)$ business traffics. We now present a real network function arrangement process produced by Sirius in Figure 11. In one of our edge clouds, there are $O(10)$ network function chains (each for one sub-business traffic) needed to be offloaded onto the single programmable gateway in this edge cloud. These business traffics contain our mainstream services, including games and streaming. These $O(10)$ network function chains are merged into a network function graph shown in Figure 11a. This graph contained 127 processing loops. Figure 11a only colors three of these loops.



(a) Merging $O(10)$ network function chains results in a network function graph containing 127 processing loops. We only color three of the processing loops in this figure.



(b) A pipelining plan output by Sirius. There are only two recirculations in this result.



(c) The pipeline after the partitioning (this figure shows the part offloaded onto the programmable ASIC).

Figure 11: A real network function arrangement in our production. Over 10 network function chains types of traffics are arranged into a single gateway. Each back node in the above figures represents a snippet of P4 code containing around ten tables. Each $\rightarrow$ denotes the execution sequence.

Figure 11b shows a pipelining result output by Sirius. This pipelining result only contains two recirculations in three subchains. Our specified throughput for all business traffics is 0.5 Tbps, so this candidate meets the specifications. More importantly, our programmers spent one week creating a traffic identification table for forwarding different business traffics to the corresponding network functions. This manually-written table occupied 15.7% of SRAM memory and 38.1% of TCAM memory of the entire data plane program. With Sirius, we got this traffic identification table within a few minutes, and the table only used 11.5% of SRAM memory and 42% of TCAM memory of the manually-written one.

Figure 11c shows a partitioning result generated by Sirius. Some of the programs in Figure 11b have been moved to the CPU. Each node in the graph shown in Figure 11 represents a snippet of code—containing around 10 match-action tables if it is implemented in P4, and each arrow denotes the execution sequence. **The entire network function arrangement process, with Sirius, was finished within 10 minutes.**

Safely and easily network function updating. Another benefit Sirius offers is to ease our network function updating across edge clouds. Different edge clouds serve different businesses; thus, the data plane programs running on different edge clouds are diverse. However, before Sirius was deployed, our programmers used to update network functions across

```
table snat_session {
  key = {
  #ifdef IPv4
    hdr.ipv4.src_ip: lpm;
  #endif
  #ifdef IPv6
    hdr.ipv6.src_ip: lpm;
  #endif
  ...
  }
  size = SNAT_SESSION_NUM;
  actions = {
  #ifdef IPv4
    snat_ipv4_rewrite;
  #endif
  #ifdef IPv6
    snat_ipv6_rewrite;
  #ifdef VXLAN
    snat_vxlan_rewrite;
  #endif
  #endif
  }}
```

Figure 12: Example code snippet using macro extensively.

edge clouds via macros in P4. On different gateways, our programmers turned on different macros, and the compiler can remove the rest code. The macro solution is, nevertheless, hard to maintain and error-prone, because the P4 programs end up with $O(10)$ different macros and $O(100)$ copies of them spread across different files. Figure 12 shows a real table definition in our edge cloud, which adds many macros to change its key and actions based on the actual needs. We can observe that multiple macros are chained and nested together. A failure event that occurred two years ago in one of our edge clouds was caused by incorrectly updating macros, since our programmers made a mistake when they updated nested macros. Sirius solves this problem. For different gateways, the programmers only need to specify network function chains in P4 and provide the correct traffic separation DB, freeing them from the "tangled" macros.

Real partitioning case. Without Sirius, even though our programmers generate a pipelining plan with the satisfied number of recirculations, it is very hard to squeeze this single pipeline into the programmable ASIC due to the limited hardware resources. Thus, our programmers used to reduce the size of some tables in some network functions that they thought would experience low volume for a while to ensure the tailored pipeline complies with the hardware constraints. After a while, when the traffic volume went back up, they moved them back and shrank some tables in another network function. The above situations occurred on almost every gateway before Sirius was built. In a recent case, our programmers spent two weeks squeezing a merged graph (with Source NAT, destination NAT, and load balancer network functions) into the programmable ASIC. They tried to reduce the size of the routing tables but still failed to compile the entire graph. Sirius helped them to automatically generate a partitioning result that guided them to move the `source_NAT_session_table` to the CPU, efficiently solving this problem. Moving `source_NAT_session_table` to the CPU does not affect our throughput, since for that edge instance, the source NAT traffic volume is low while the session table sits in a critical position in the network function pipeline. Assigning it to the CPU not only frees up a lot of memory resources but also significantly shortens the length of the table dependency chain [13], which creates more headroom and makes the program easier to compile.

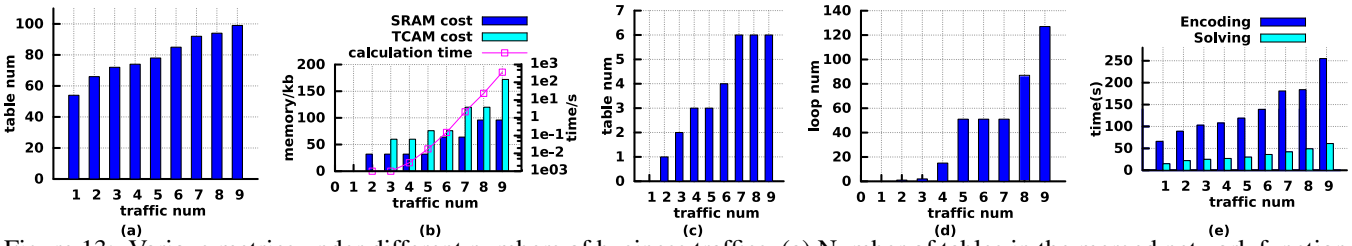


Figure 13: Various metrics under different numbers of business traffics. (a) Number of tables in the merged network function graph. (b) Memory cost and execution time of constructed traffic identification tables. (c) Number of traffic identification tables. (d) Number of loops in the merged graph. (e) Time to encode and solve the SMT formula.

7.3 Lessons and Discussions

We now share our lessons in using Sirius in our edge clouds, and also discuss the limitations of Sirius and open questions.

Can any of the pipelining candidates directly comply with the hardware constraints? Our one-year experience shows that none of the pipelining plan candidates can directly comply with the programmable ASIC's constraints. In other words, the partitioning between the programmable ASIC and the CPU is a must for the network function arrangement in our edge clouds. This situation results from (1) the limited hardware resources in programmable ASICs, and (2) our large-scale production programs.

Can a pipelining plan with more processing loops produce a better partitioning result? This may happen. We selected pipelining plan candidates that meet the specified throughput, and then partitioned these candidates. In other words, the pipelining plans that violate the specification are ineligible to join the partitioning phase, because they cannot be better than the pipelining candidates meeting the specification, no matter how to partition them. However, for the pipelining candidates, a candidate with more processing loops might be partitioned to a result with less CPU usage than another candidate with fewer processing loops.

Duplicating the overlapping network functions to remove processing loops is impractical. We now use the recirculation to remove processing loops. Another option is to duplicate the overlapping network functions to remove the loops. While such a solution, in principle, works and does not sacrifice the throughput, it is impractical due to the limited hardware resources. Since a network function needs about 5-10 stages, additional 5-10 stages for one processing loop is too expensive to the precious hardware resources.

Optimization target. Sirius currently minimizes the total number of tables assigned to the CPU, which is not always the best optimization target. For example, in some cases, our programmers want to pin some tables or network function chains onto the programmable ASIC to achieve consistency or low processing latency. Sirius can support such cases by only searching through candidates that satisfy the requirement.

Gateways in edge clouds v.s. Gateways in data center networks. We have equipped gateways with programmable ASICs in both edge clouds and data center networks. Compared with the data center case, deploying programmable data planes in the edge clouds is much more challenging.

First, each gateway in a data center network needs to hold only one network function, unlike the edge cloud, where a gateway needs to hold a large number of network functions. This is because there are $O(100)$ gateways in a data center network in our global network, and multiple network functions can be horizontally distributed across these gateways without squeezing all of them into a single gateway [16]. On the contrary, each edge cloud only contains two gateways (for redundancy) since edge clouds are typically deployed in the rented cheap, small-size machine rooms close to end users. Each gateway in our edge clouds needs to hold $O(100)$ network functions, and squeezing so many network functions into a programmable ASIC is very hard due to limited hardware resources. For the same reason, it is impossible to deploy multiple groups of gateways in the edge clouds.

Second, network functions in the edge clouds are logically more complex. In the edge clouds, the gateways process packets from over 10 types of business traffics. On the contrary, gateways in the data center just forward packets to the corresponding switches. We typically developed the P4 program for data center network gateways by just tailoring `switch.p4`.

Partitioning across switching ASIC, smartNIC, and CPU. Offloading network functions to smartNIC, in principle, can alleviate programmable ASIC hardware resources issue while maintaining the throughput, It can replace CPU and reduce the overall cost of the gateway. We are developing the next-generation gateway that replaces CPU with SmartNIC. We are also working on building a partitioning approach to distribute the code across ASIC, smartNIC, and CPU in order to balance the trade-off between performance and cost. While FlightPlan [23] has presented a good effort toward this direction, deploying such a system in production remains challenging.

8 Evaluation

Our evaluation mainly focuses on presenting the scalability of each component in Sirius. We chose 9 different business traffics, added them one by one, and recorded the time of execution and other metrics we were interested in. All experiments were performed on a server with a 2.5GHz CPU and

768GiB RAM.

Figure 13(a) shows the program scale. There are two observations. Firstly, different business traffics had many tables in common. Because the number of tables only doubled when we added 8 more business traffics. Secondly, the difference between different business traffics is similar, as the number of tables grows gradually as the traffic number increases.

Traffic identification table construction. We recorded the SRAM and TCAM usage under different numbers of network functions, and the time Sirius took to solve the problem. Figure 13(b) shows the result. Both types of memory increased as the number of traffics increased, which showed the overhead of distributing flows to different network functions. Also, the time it took to compute the minimal traffic identification table increased dramatically as the number increased. This is because Sirius had to try more combinations to find the optimal result. When there were 9 business traffics, Sirius took 371 seconds to construct the distribution tables. We also recorded the number of tables constructed in Figure 13(c). We observed that even though the memory increased, the number of tables did not always increase. This is because Sirius chose multiple rules that shared the same match field so that they could share the same table.

Pipelining. In the pipelining process, the algorithm Sirius used to solve the set covering problem is efficient enough to finish all experiments within a second. So we mainly present the number of processing loops in the network function graph as more business traffics are added. Figure 13(d) shows the result. The number of processing loops increases quickly as the business traffics become more complicated, which also demonstrates the programmers' workload before Sirius is deployed. Programmers had to plan carefully to make sure each business traffic visit functions in its desired order.

We also noticed that when adding the 6th and 7th business traffic, the number of loops stayed the same. This is because those business traffics are similar, and they shared the same code snippet that was in the loop.

Partitioning. In the partitioning phase, we mainly evaluated the time for Sirius to encode the SMT formula and for the SMT solver to solve it. The result is shown in Figure 13(e). Both the encoding and solving time increase as the number of business traffics increases. We can see that the encoding time grows faster than linear because the encoding is built atop the network function pipeline, whose complexity grows faster than linear as the business traffic is added. Solving the formula is faster than encoding it. This is because Sirius leverages the constraint encoding optimizations proposed by Cetus [13]. Overall, the partitioning finishes within minutes.

9 Related Work

Partitioning hardware code to CPU. Closed to Sirius's partitioning goal, Gallium [27] automatically translates a part of software-version middlebox programs into a P4 program running on a programmable ASIC. Gallium does not support recirculation modeling and mainly targets a specific switch-CPU architecture. In addition, Sirius's partitioning goal is also different from Gallium's. FlightPlan [23] deployed P4 programs across the switch, FPGA, and CPU to benefit bandwidth and heterogeneity. FlightPlan relies on programmers to explicitly split the program and profile each code block's performance on each platform. We cannot apply FlightPlan, since the above requirements do not hold in our scenario. Some of the prior work targeted partitioning between the SmartNIC and the CPU, such as iPipe [14], Clara [15], and Floem [17]. These efforts are quite different from Sirius's focused goal and assumptions. Dejavu [25] leverages recirculation to compose multiple network function chains to a single ASIC but it cannot handle the three challenges Sirius solves.

Compiler for programmable data planes. The state-of-the-art compiler systems for P4 [3, 5, 9, 10, 19, 22] aim to optimize resource usage in programmable ASICs or simplify programmers' tasks on expressing their coding intent. For example, P4All [9, 10] optimizes resource usage by leveraging reusable data structures. P4visor [29, 30] optimizes resources by merging redundant code fragments (*e.g.*, header parser and tables). μP4 [22] and P4 Weaver [3] propose P4 modular programming to write P4 code from scratch or in an incremental way.

Extend switch memory with host memory. TEA [12] addresses the memory constraint issue on the programmable ASIC by extending ASIC memory with the DRAM on server machines. Even though memory occupation is one of our major concerns, as shown in Figure 2 and Cetus [13], our long network function chain is another major concern. TEA reduces memory footprint by introducing more computation logic in the programmable ASIC. This worsen our network function chain dependency issue and does not solve our problem.

10 Conclusion

This paper has shared our design of Sirius, and our experience with Sirius in our edge clouds after one year of use. Sirius is the first system capable of automating the network function arrangement problem. Compared with manual solutions to the network function arrangement problem, Sirius effectively decreased our workload for deploying and updating our edge clouds by three orders of magnitude from weeks to minutes.

This work does not raise any ethical issues.

Acknowledgement

We thank our shepherd, Ming Liu, and NSDI'24 reviewers for their insightful comments. Jiamin Cao was supported by Alibaba Group through Alibaba Research Intern Program. Ennan Zhai is the corresponding author.

References

[1] ASAI, H. Palmtrie: A ternary key matching algorithm for ip packet filtering rules. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2020), CoNEXT '20, Association for Computing Machinery, p. 323–335.

[2] EADES, P., LIN, X., AND SMYTH, W. F. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters 47*, 6 (1993), 319–323.

[3] FATTAHOLMANAN, A., BALDI, M., CARZANIGA, A., AND SOULÉ, R. P4 weaver: Supporting modular and incremental programming in P4. In *Symposium on SDN Research (SOSR)* (2021).

[4] FATTAHOLMANAN, A., BALDI, M., CARZANIGA, A., AND SOULÉ, R. P4 weaver: Supporting modular and incremental programming in p4. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (2021), pp. 54–65.

[5] GAO, J., ZHAI, E., LIU, H. H., MIAO, R., ZHOU, Y., TIAN, B., SUN, C., CAI, D., ZHANG, M., AND YU, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM (SIGCOMM)* (2020).

[6] GAO, X., KIM, T., VARMA, A. K., SIVARAMAN, A., AND NARAYANA, S. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[7] GAO, X., KIM, T., WONG, M. D., RAGHUNATHAN, D., VARMA, A. K., KANNAN, P. G., SIVARAMAN, A., NARAYANA, S., AND GUPTA, A. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)* (2020).

[8] HANCOCK, D., AND VAN DER MERWE, J. HyPer4: Using P4 to Virtualize the Programmable Data Plane . In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2016).

[9] HOGAN, M., FEIBISH, S. L., ARASHLOO, M. T., REXFORD, J., AND WALKER, D. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2022).

[10] HOGAN, M., FEIBISH, S. L., ARASHLOO, M. T., REXFORD, J., WALKER, D., AND HARRISON, R. Elastic switch programming with P4All. In *19th ACM Workshop on Hot Topics in Networks (HotNets)* (2020).

[11] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 517–530.

[12] KIM, D., LIU, Z., ZHU, Y., KIM, C., LEE, J., SEKAR, V., AND SESHAN, S. TEA: enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM (SIGCOMM)* (2020).

[13] LI, Y., GAO, J., ZHAI, E., LIU, M., LIU, K., AND LIU, H. H. Cetus: Releasing P4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2022).

[14] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartNICs using iPipe. In *ACM SIGCOMM (SIGCOMM)*. 2019.

[15] NARAIN, S., LEVIN, G., MALIK, S., AND KAUL, V. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage. 16*, 3 (2008), 235–258.

[16] PAN, T., YU, N., JIA, C., PI, J., XU, L., QIAO, Y., LI, Z., LIU, K., LU, J., LU, J., ET AL. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 194–206.

[17] PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODIK, R., AND ANDERSON, T. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[18] SCHOLZ, D., STUBBE, H., GALLENMÜLLER, S., AND CARLE, G. Key properties of programmable data plane targets. In *2020 32nd International Teletraffic Congress (ITC 32)* (2020), pp. 114–122.

[19] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 15–28.

[20] SONCHACK, J., LOEHR, D., REXFORD, J., AND WALKER, D. Lucid: A language for control in the data plane. In *ACM SIGCOMM (SIGCOMM)* (2021).

[21] SONI, H., RIFAI, M., KUMAR, P., DOENGES, R., AND FOSTER, N. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 329–343.

[22] SONI, H., RIFAI, M., KUMAR, P., DOENGES, R., AND FOSTER, N. Composing dataplane programs with μp4. In *ACM SIGCOMM (SIGCOMM)* (2020).

[23] SULTANA, N., SONCHACK, J., GIESEN, H., PEDISICH, I., HAN, Z., SHYAMKUMAR, N., BURAD, S., DEHON, A., AND LOO, B. T. Flightplan: Dataplane disaggregation and placement for P4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2021).

[24] WINTERMEYER, P., APOSTOLAKI, M., DIETMÜLLER, A., AND VANBEVER, L. P2GO: P4 profile-guided optimizations. In *The 19th ACM Workshop on Hot Topics in Networks (HotNets)* (2020).

[25] WU, D., CHEN, A., NG, T. S. E., WANG, G., AND WANG, H. Accelerated service chaining on a single switch ASIC. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[26] ZHANG, C., BI, J., ZHOU, Y., DOGAR, A. B., AND WU, J. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), pp. 1–9.

[27] ZHANG, K., ZHUO, D., AND KRISHNAMURTHY, A. Gallium: Automated software middlebox offloading to programmable switches. In *ACM SIGCOMM (SIGCOMM)* (2020).

[28] ZHENG, P., BENSON, T., AND HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (2018), pp. 98–111.

[29] ZHENG, P., BENSON, T., AND HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2018).

[30] ZHENG, P., BENSON, T. A., AND HU, C. Building and testing modular programs for programmable data planes. *IEEE J. Sel. Areas Commun. 38*, 7 (2020), 1432–1447.