Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling

Yifan Li[‡,†], *Jiaqi Gao*[†], *Ennan Zhai*[†], *Mengqi Liu*[†], *Kun Liu*[†], *Hongqiang Harry Liu*[†]
[‡]*Tsinghua University* [†]*Alibaba Group*

Abstract

Programmable switches are widely deployed in Alibaba's edge networks. To enable the processing of packets at line rate, our programmers use P4 language to offload network functions onto these switches. As we were developing increasingly more complex offloaded network functions, we realized that our development needs to follow a certain set of constraints in order to fit the P4 programs into available hardware resources. Not adhering to these constraints results in *fitting issues*, making the program uncompilable. Therefore, we decide to build a system (called Cetus) that automatically converts an uncompilable P4 program into a functionally identical but compilable P4 program. In this paper, we share our experience in the building and using of Cetus at Alibaba. Our design insights for this system come from our investigation of the past fitting issues of our production P4 programs. We found that the long dependency chains between actions in our production P4 programs are creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of our fitting issues. Guided by this finding, we designed the core approach of Cetus to efficiently synthesize a compilable program by shortening the lengthy dependency chains. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from O(day) to O(min)). In this paper we share several real cases addressed by Cetus, along with its performance evaluation.

1 Introduction

Programmable switches allow network programmers to use P4 language to offload network functions to data planes, enabling these functions to process packets at line rate. As one of the largest global service providers, Alibaba has widely deployed programmable switches in its edge networks [20, 27]. By Jan 2021, we have built $O(100)$ PoP (point of presence) nodes and $O(1000)$ edge sites in total, and the majority of them have employed programmable switches to implement a group of network functions, including firewall, DDoS defense, and load balancer. Figure 1 shows an example of the architecture of network functions within a single programmable switch in our edge networks. In this architecture, our programmers offload multiple network functions to a single programmable switch, enabling these network functions to process packets at Tbps speeds and saving CPU resources on the end-servers in edge networks.

While our business significantly benefits from the deployment of programmable switches, nevertheless, we still encounter a tough problem. Our P4 program development—*e.g.*, implementation of new network functions and update of the existing network functions via P4—needs to take into account the various constraints of programmable switching ASICs; neglecting these constraints often results in programs that cannot fit on the hardware and hence cannot compile. We call this problem as *fitting issue*.

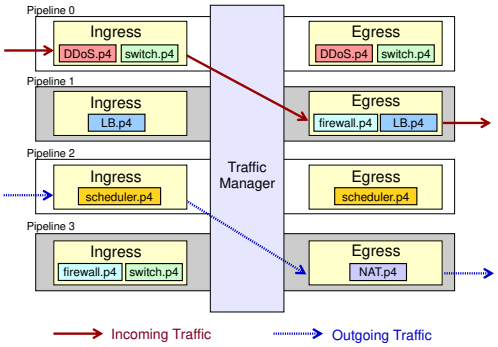


Figure 1: A gateway P4 program example deployed in Alibaba's edge network. In our edge network scenario, our programmers put various network functions in a single switch.

Fitting a P4 program is hard to our programmers, because (1) programmable switching ASICs have various hardware resources, each with unique size and constraints, and (2) resources are sometimes correlated, reducing the resource A usage of a program coming at the cost of increasing the usage of resource B. Our programmers, therefore, usually fall into time consuming trial and error program "reshaping" cycles, significantly delaying their development time. On the other hand, it is impractical to require our programmers to learn all hardware constraints.

Alibaba therefore decided to build a system (called Cetus) that automatically converts an uncompilable P4 program P into a functionally identical but compilable P4 program P'.

State of the art. Existing work falls into two categories. On the one hand are systems that compile a high-level abstraction to generate optimized P4 programs [10, 13, 14, 25, 30]. Although they offer good resource optimizations, we found these solutions may not be effective in our specific scenario. For example, P4All [13, 14] optimizes the resource usage among network functions by explicitly leveraging reusable data structures (*e.g.*, bloom filters and key-value stores); however, the network functions within our production P4 programs do not share these data structures, invalidating this optimization in our case. In addition, our programmers are reluctant to use

an extension of P4 such as explicitly specifying some data structure to optimize via `objective` in P4All. Another state-of-the-art system, Lyra [10], merges the tables that have no dependencies with each other in order to optimize the resource usage; however, we found that merging tables while keeping the original dependencies is not enough to enable our production P4 programs to fit into the programmable ASICs. On the other hand, existing efforts like Chipmunk [11, 12] and Domino [24] improve P4 compiler to synthesize optimized switch binary code, which is different from our goal of generating optimized P4 programs.

Our approach: Cetus. This paper shares our experience in the building and using of Cetus at Alibaba. We first investigated our production P4 programs and their past fitting issues, in order to derive insight for our solution design. We found that the **long** dependency chains between actions in our production P4 programs were creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of fitting issues.

Guided by the above finding, we designed Cetus. For a given P4 program P, Cetus automatically merges tables to fit into fewer stages by removing dependencies between tables, thus shortening the long dependency chains (§5). Because such a method may generate many table merging options (called candidates), we propose an approach, called constraint-based filter & optimizer (§6), to drop the candidates that do not satisfy hardware resources (including memory size, PHV, and crossbar) or constraints, and then select the best one as P'. Designing such a filter & optimizer approach is non-trivial due to two challenges: (1) the large formula encoding each candidate may result in state explosion, and (2) large solution searching space in each candidate will cause long solving time. We propose PHV sharing encoding (§6.1) and two-step solving (§6.2) to address the above two challenges, respectively. With P' in hand, Cetus automatically generates a set of control plane APIs for P' to enable P' to be deployed seamlessly (§7).

Finally, we share several representative real cases addressed by Cetus (§8), along with its performance evaluation (§9). We have been using Cetus in production for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from O(day) to O(min)).

2 Preliminary: Programmable Data Plane

We use ϒ to denote the name of programmable switching ASICs of Vendor A.[1] Our programmers compile P4 programs via ϒ compiler. ϒ chip is a physical implementation of *Protocol Independent Switch Architecture* (or PISA). ϒ chip's ingress and egress consist of 12 stages, respectively. All of these stages are identical, in terms of compute units, memory types, and memory capacities.

[1] We omit the vendor name and ASIC name for the confidentiality.

2.1 Hardware & Constraints of ϒ Chip

Hardware resource. ϒ chip contains various hardware resources, and each of them has unique size and characteristic. We are mainly focused on the following hardware resources:

- **Pipeline stages.** The packet processing pipeline consists of a fixed number of individual stages. A P4 program does not compile if it takes more than 12 stages in an ingress or egress pipeline in ϒ chip.
- **Packet header vector (PHV).** The PHV is a "bus" that carries information (from packet fields and per-packet metadata) between stages. PHV cannot carry more data than its total width. See §6.1 for more PHV details.
- **Memory.** Memory resources mainly contain SRAM and TCAM. SRAM and TCAM are around tens of Megabytes in capacity. The memory resources are equally split and attached to each stage so that each stage can only access its local memory resources.
- **Crossbar.** In each stage, the crossbar extracts fields from the PHV and sends them to the match and action units for computation. Crossbar has a size limit, so the total number of bytes assigned to a stage's crossbar should not exceed this limit.

Hardware constraints. The hardware constraints, in this paper, refer to both the hardware resource characteristics (*e.g.*, in ϒ chip, memories are stage local, and memory can be accessed no more than once per packet), and the mappings between the P4 program elements and hardware resources (*e.g.*, a P4 table's keys should be stored in SRAM or TCAM memory, and a packet header field should be mapped into one or multiple cells in the PHV). Understanding these hardware constraints is crucial to programming on the ϒ chip.

To successfully compile a P4 program via ϒ compiler, this program must not exceed the size of each hardware resource and comply with all constraints of ϒ chip.

Fitting a P4 program in our practice. Our production P4 programs typically pack as many functions and modules as possible, which may overuse hardware resources or violate the hardware constraints, resulting in the fitting issues. When this happens, our programmers have to 'reshape' the programs to fit into the programmable ASIC. Such a reshaping process is program specific. Our programmers often spend a significant amount of time reshaping our P4 programs in order to comply with the hardware resources and constraints.

2.2 Dependencies between Tables

A P4 program is a collection of match-action tables chained together by branching conditions. In each table, at most one action can be applied according to the match result. For a given group of actions, if there is no read-write or write-write dependency among these actions, they could be placed within the same stage. On the contrary, for example, if action i_1 uses (reads or writes) a value generated by action i_2, then i_1 must

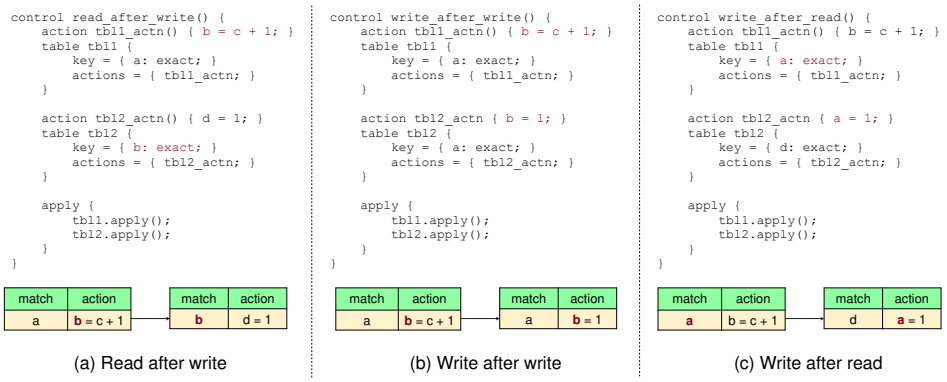
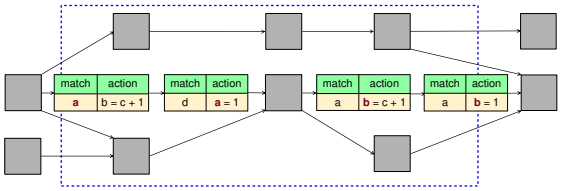
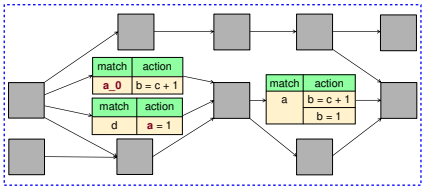


Figure 2: Three types of dependencies between actions in our production P4 programs.



(a) The match-action DAG for the original P4 program



(b) The match-action DAG for P4 program after optimization

Figure 3: Examples for match-action DAGs. Rectangles represent tables. The blue dashed frame represents the architecture of ϒ chip. The blue dashed frame's length and width represent the usages of stage and memory, respectively, in ϒ chip. (a) shows a match-action DAG representing a given P4 program P. P does not fit in ϒ chip. (b) is a match-action DAG representing P' that tweaked from P, which is compilable.

be placed in a stage after the stage of i_2 in the PISA architecture. In our production P4 programs, we are mainly focused on three types of dependencies: read after write, write after write and write after read[2]. Figure 2 shows their examples. The tables, in Figure 2(a), (b), and (c), are not allowed to be *directly* placed within the same stage; otherwise, the programs' function logic is changed.

Match-action DAG. By tracking dependencies between actions, we can represent a P4 program in the form of a match-action directed acyclic graph or *match-action DAG*. Figure 3(a) presents such a match-action DAG.

Diameter of a match-action DAG. The total number of stages occupied by a P4 program P cannot be less than the *diameter* of the match-action DAG representing P. The diameter of a match-action DAG G is: the number of tables in the longest dependency chain (*i.e.*, the dependency chain containing the highest number of tables) in G. For example

[2]We explain why write after write dependency is necessary in §5.1

P4 Programs	Network Functions	Diameter		Head, Tail Memory Percentage
		Ingress Pipeline	Egress Pipeline	
Edge vSwitch	VXLAN encapsulation VXLAN decapsulation Controlling the flow between CPU and data plane Traffic statistic IP packet forwarding ACL	9	3	14.73%, 3.32%
CDN	Load balancing Controlling the flow between CPU and data plane Scheduling IP packet forwarding DDoS defense ACL	10	5	0.87%, 5.04%
Edge Gateway	VXLAN packet forwarding Traffic limit Load balancing ACL	8	7	0.01%, 0.86%

Figure 4: Our production P4 programs and their involved network functions as well as their diameters. These three programs have been deployed on almost all the programmable switches in our edge networks.

in Figure 3(a), the diameter is 7, because there are 7 tables in the longest dependency chain of the DAG. The diameter in Figure 3(b) is 5. Thus, we can say that the diameter of a match-action DAG (representing P) must be $\leq$ the number of stages, if P compiles.

3 Key Findings & Solution Intuition

In order to release our programmers from trial and error program-reshaping cycles, we need to understand the root causes resulting in fitting issues during the development of our production programs, thus exploring insights for our solution design. Specifically, we selected three mainstream P4 programs (listed in Figure 4) in our production, which were deployed in almost all the edge switches in Alibaba edge networks. We then selected all fitting issues (of these three programs) that took our programmers more than one hour to resolve, and manually analyzed how they were fixed.

We classified our analysis results into two groups. (1) **Group A:** About 80% of fitting issues were resolved by eliminating or reducing dependencies between tables (*e.g.* by reordering or merging them) that allowed us to take advantage of the parallel nature of the switch architecture. (2) **Group B:** 20% issues were resolved by fixing hardware resources and constraints that programmers were not aware of such as PHV

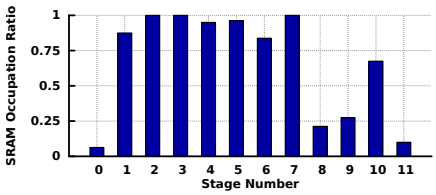


Figure 5: SRAM usage of Edge Gateway program.

allocation and stateful ALUs. We now analyze the principles behind Group A (§3.1) and Group B (§3.2).

3.1 Key Findings from Group A

We investigated why rearranging tables can resolve the fitting issues in this group. We found that all of these efforts (*e.g.*, reordering and merging tables) implicitly shortened the P4 programs' diameters. For example, in one of the cases, our programmer unwittingly merged two tables by changing dependencies between their actions (as shown in Figure 7(a) example), and then found that the program compiled. While this programmer did not know the fundamental reason (*i.e.*, shortening the diameter), he succeeded after multiple reshaping cycles.

Observation 1: Diameter is long in our production. Why shortening the diameter can resolve the fitting issue? We found that the match-action DAG representing each of these three P4 programs had long diameters. Given that Υ chip provides 12 stages of match-action units, a long diameter should be reduced in order to make programs fit on Υ chip. As shown in Figure 3(a), blue dashed frame's length and width represent the usages of stage and memory, respectively, in Υ chip. The program's diameter in Figure 3(a) is too long to comply with the stage resource size.

The long diameter results from the large number of packet processing operations required by our diverse edge services. In particular, each of our P4 programs not only needs to insert various metadata into the different types of packet headers, but also filters or forwards them according to a number of service needs. For example, an input packet is first encapsulated with VXLAN, then forwarded based on some condition, next mirrored for traffic statistics and finally checked by ACL as well as distributed by the ECMP. Figure 4 details these three P4 programs' diameters and their involved network functions. All programs shown in Figure 4 have at least a diameter of 8 in ingress, which means they occupy at least 8 stages in the ingress pipeline. It is therefore highly possible to result in fitting issues in Υ chip when new tables are added.

Observation 2: Many available memory resources. We also found that shortening the diameter by tweaking tables, in principle, increases the usage of memory within individual stages, as shown in Figure 3(b). Why did this memory-for-stage method work in our production? We found that both ends of the match-action DAG (tables with 0 in-degree or out-degree) use much less memory, offering flexibility for table tweaking.

At the beginning of the pipeline, our programs need to perform checking and pre-computations such as packet validation, link aggregation group checking, pre-computing hashes, and setting flag based on header's validity; at the end of the pipeline, our programs finalize the packet processing based on the previous matching results, including marking header fields, dropping packets, and encapsulations. All these operations can be easily done in parallel, while at the same time they do not require a lot of table entries; thus, much available memory remains. Figure 4 shows the percentage of memory that both ends of DAG occupy compared with the entire program. If the memory is distributed evenly across the DAG, both ends of the DAG should occupy around 10% of memory each. Figure 5 shows the SRAM occupation ratio per stage of Edge Gateway program (*i.e.*, the third program in Figure 4). We observed that stage 0 and 11 only used less than 10% of memory. The other two programs also follow the same phenomena.

We also observed much available memory in the middle of the pipeline. Figure 5 shows tables at stage 8 and 9 take only 25% of memory. Similar phenomena also occurred in the rest of the two P4 programs listed in Figure 4. This is because, in a network function chain, we typically have a few tables that are small but critical such as a table inserting a mainstream service-shared DSCP value into the packet header as metadata. Such a table (called T) must have (read-write or write-write) dependency relationships with the tables before and after T.

Summary. We now understand that our programmers unwittingly shortened the diameter of their programs by trial and error table (dependency) tweaking, luckily making their programs compile. Examples in Figure 3(a) and (b) illustrate such an intuition. We therefore derive the following key finding.

> **Finding 1:** Long dependency chains between actions in our production P4 programs make the developed programs hard to fit into the programmable ASIC. We thus need to remove dependencies on the "longest path" of DAG to change the original "long, narrow" DAG to a "short, fat" DAG, as shown in Figure 3, in order to enable our developed programs to compile.

3.2 Key Findings from Group B

Fitting issues in Group B were caused by the violation of chip-specific resource size and hardware constraints. For example, because our programmers ignored the size of an individual stage, the program they wrote required the compiler to assign more DRAM within one stage than allowed (otherwise the dependency constraint is violated), resulting in a fitting issue; the same issue also happened for other resources such as hash units. There is no pattern to follow among these root causes. But we noticed that some of the issues in Group B were caused by same constraint violation. This means that our

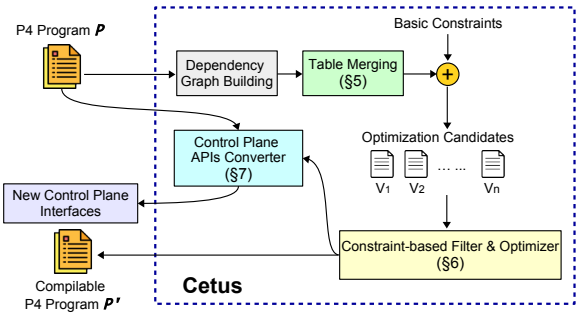


Figure 6: Cetus's workflow overview.

programmers failed to learn or remember the fitting issues that they have ever fixed. We thus derive the following finding.

> **Finding 2:** Although it might be hard for our programmers to learn all chip-specific resource size and constraints, we should avoid the fitting issues—resulting from the unfamiliarity with the resource size and constraints—that we have encountered before.

3.3 Our Solution Idea

Based upon our above two findings, we design the core approach of our solution, which includes the following three steps. First, for a given P4 program P, we automatically merge tables to fit into fewer stages by removing dependencies between actions, in order to shorten the long diameter of DAG representing P (driven by Finding 1). Such an approach would generate many candidate results. Second, we encode hardware resource size and hardware constraints as many as we know in our system's backend DB to ensure that the synthesized program complies with all already-known resource size and constraints (driven by Finding 2). Finally, we check each candidate with the encoded constraints, selecting the most optimal one.

Why the state of the art does not help? Existing systems (*e.g.*, Lyra [10] and P4All [13, 14]) are unable to offer such a level of program optimization. Specifically, Lyra can only merge tables without dependencies. In other words, Lyra cannot merge two tables by removing dependencies between the tables; thus, Lyra is unable to shorten the diameter of the given DAG. P4All optimizes programs by reusing common data structures. In our programs (shown in Figure 4), however, the tables on the diameter do not share any data structure, invalidating P4All's assumption.

4 Cetus's Workflow Overview

We build Cetus, a synthesis system that automatically converts an uncompilable P4 program P into a functionally identical but compilable P4 program P'.

Figure 6 presents Cetus's workflow that consists of the following main phases.

- First, given a P4 program P, Cetus generates a match-action DAG by analyzing read-write and write-write dependencies in P. Then, Cetus introduces a table merging approach (§5) to shorten the diameter of the generated DAG by removing dependencies between tables. There could be many potential table merging cases. We drop the cases that violate basic hardware constraints (*e.g.*, memory size), obtaining a group of candidate programs.
- Second, we propose a constraint-based filter and optimizer (§6) to check each candidate individually with already-known constraints, selecting the most optimal one as P'.
- Finally, Cetus automatically generates a set of control plane APIs for P' to enable P' to be deployed seamlessly (§7).

5 Table Merging by Dependency Removal

Cetus proposes a table merging approach to shorten the diameter by removing dependencies. Intuitively, the purpose of the table merging module is to tweak P to fit into the architecture of Υ chip. This approach includes several primitives to merge tables for different types of dependencies. This section first introduces these primitives (§5.1), and then describes the entire solution (§5.2).

5.1 Dependency Removal Primitives

We design several dependency removal primitives in terms of dependency types, including write-after-write, write-after-read and read-after-write dependencies (shown in Figure 2). Each of the primitives takes two tables as input and returns one or two tables that can be put within one single stage. The purpose of these primitives is to reduce the number of used stages by increasing other resources' overhead such as PHV and memory.

Symbols. We define the following notations: table t has n_m match fields $\{m_{t1}, ..., m_{tn_m}\}$, each field m_{ti} has w_{ti} bits in width and its match type is p_{ti}, which can be *exact*, *ternary*, *etc*. It also has n_a actions $\{a_{t1}, ..., a_{tn_a}\}$. If one table has no default action, we add an empty action as the default. Table t has l_t entries. Let P_t be the action parameters' total bit width, then table t's total memory usage is $l_t(\sum_{i=0}^{n_m} w_{ti} + P_t)$.

Write-after-write (WAW) dependency. WAW dependency happens when one table t_1 contains an action that writes the value written by another table t_2. For example, in Figure 2(b), table `tbl2`'s action *tbl2_actn* writes variable b, which is previously modified by table `tbl1`[3]. Since two actions are not allowed to write to the same data in a PHV word concurrently, one cannot place them in the same stage. It is also impossible to reorder them since the program's correctness is violated.

This primitive removes WAW dependency by merging the two tables into a new table t'. The merged table t' enumerates both tables' all action combinations. The primitive works as follows, and Figure 7(a) shows an example.

[3] We cannot remove `tbl1` because a packet can hit `tbl1` but miss `tbl2`.

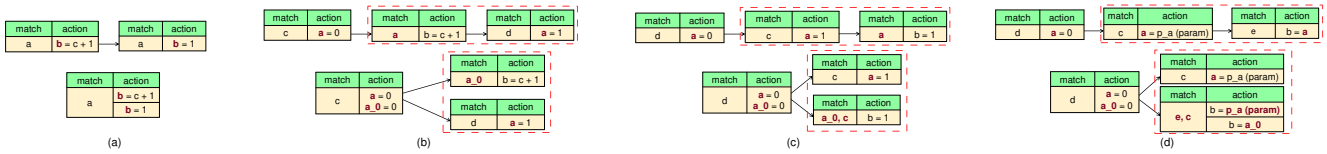


Figure 7: Examples for different dependency removal primitives. (a) WAW (b) WAR (c) RAW-match (d) RAW-action

- Merge the match fields of the two tables and generate new match fields $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\}$.
- Generate all possible combinations of the two tables' actions $\{(a_{t_1 1}, a_{t_2 1}), (a_{t_1 2}, a_{t_2 1}), ..., (a_{t_1 n}, a_{t_2 n})\}$
- Merge each pair of actions into one by appending the statements in the second action after the first one.
- When a merged action has two statements that write the same value, one from t_1, one from t_2, we keep the latter one.

Memory usage. Since the two tables hit and miss independently, the merged table should include all four possibilities. Thus, unless two tables have identical match fields, table t' uses ternary match field types and is deployed in the TCAM memory. In total, there are $(l_{t_1}+1)(l_{t_2}+1)-1$ entries. The total memory usage of table t' is $l_{t_1} l_{t_2} (\sum_{i=0}^{n_{t_1 m}} w_{t_1 i} + P_{t_1} + \sum_{i=0}^{n_{t_2 m}} w_{t_2 i} + P_{t_2})$.

Write-after-read (WAR) dependency. When one table t_2 writes the variable read by t_1, WAR dependency happens. For example, in Figure 2(c), the table `tbl2`'s action *tbl2_actn* writes variable a, which is table `tbl1`'s match fields. Again, we cannot reorder these two tables; however, PISA architecture allows t_2 to be deployed alongside t_1. When t_1 occupies multiple stages, t_2 can only share t_1's last stage and not earlier. WAR dependency does not necessarily increase the total number of stages of a program directly, but it sets a "barrier" and pushes other tables to later stages. For example, in Figure 2(c), if we have a third table `tbl3` that reads variable a after table `tbl2`, then it has to be deployed after table `tbl1`, even though there is no dependency between `tbl1` and `tbl3`.

For WAR dependency, let x be the shared variable. We have table t_1 reads x and table t_2 writes it. To remove WAR dependency, we create a new copy of the shared variable x' and modify t_1 so that it reads x' instead of x. The primitive works as follows, and Figure 7(b) shows the example.

- Find the table where x is last written. If such a table exists, copy the action that writes x and modifies it to write x'. If no such table exists, such as x is a header, then we assign the value of x' in the parser.
- Modify table t_1's match and action list so that it reads x'.

Memory usage. This primitive does not create a new table and the memory usage is kept the same. It may introduce PHV overhead since it creates a new variable.

Read-after-write (RAW) dependency. Read-after-write dependency happens when one table (t_2) reads the value created by another one (t_1). For example, in Figure 2(a), table `tbl2`'s match fields read the value written by table `tbl1`'s action. The dependency can also happen when the value is read in the action field. Same as the WAW dependency, two tables with RAW dependency between them have to be placed in different stages and cannot be reordered.

This primitive removes the RAW dependency by summarizing the primitives used in WAW and WAR dependency: we first create a new table t' that summarizes the match fields of both tables and replace t_2, and then we adopt WAR dependency removal primitive to remove the dependency between t_1 and t'.

Let $x = f(\boldsymbol{v_1})$ be action in t_1 that modifies shared variable x. In Figure 2(a), $\boldsymbol{v_1}$ is $\{c, 1\}$. Assume the action is executed when table t_1 matches value $\boldsymbol{v_2}$, then after applying table t_1, x's value is:

$$x = \begin{cases} f(\boldsymbol{v_1}) & \text{if } (m_{t_1 1}, m_{t_1 2}, ..., m_{t_1 n}) = \boldsymbol{v_2} \\ x_0 & \text{otherwise} \end{cases} \quad (1)$$

where x_0 is the value of x before applying table t_1. The key of the dependency removal primitive is to encode enough information in a new table t' to compute variable x without using the result in t_1. Equation 1 shows that x depends on three sets of variables $\boldsymbol{v_1}, \boldsymbol{v_2}, x_0$. We can learn $\boldsymbol{v_2}$ from entries in table t_1. x_0 is created before t_1, so we borrow the primitive used in WAR dependency removal and create a new copy of variable x. So our challenge is reduced to understanding $\boldsymbol{v_1}$.

Theoretically, since variables in $\boldsymbol{v_1}$ have fixed lengths, we can enumerate all possibilities. However, this would lead to too much memory overhead. As a result, we only remove RAW dependency when we can infer values in $\boldsymbol{v_1}$ easily, such as when all of them are numbers or assigned to numbers directly. In Figure 2(a), $\boldsymbol{v_1} = \{c, 1\}$. If we can infer the value of c, then we can merge `tbl1` and `tbl2`, otherwise, we cannot. Cetus removes dependency differently depending on whether table t_2 reads variable x in match or action part. If table t_2 reads x in the match fields, the primitive works as follows, and Figure 7(c) shows the example tables and merged result.

- Create a copy of variable x through the method introduced in the WAR dependency removal primitive, let the copy be x_0.
- Merge the match fields of the two tables, remove x, and generate new match fields $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\} - \{x\} + \boldsymbol{v_1} + \{x_0\}$.
- Remove constants from the match field. For example when $\boldsymbol{v_1}$ or x_0 is fixed.

	RAW	WAW	WAR
Direct stateful objects	N	N	Y
Normal & not directly involved	Y	Y	Y
Normal & directly involved	N	Y	Y

Table 1: Cetus applies primitives to different cases.

- Generate a new table t' with the new match fields. Copy table t_2's action field to the table t'.

If the table t_2 reads x in the action field, we need to encode both branches in Equation 1 and duplicate actions that read x. The primitive works as follows, Figure 7(d) shows the example tables and merged results.

- Create a copy of variable x through the method introduced in the WAR dependency removal primitive, let the copy be x_0.
- Merge the match fields of the two tables and generate new match field $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\} + \boldsymbol{\nu_1}$.
- Remove constants from the match fields.
- For each action $a_{t_2 i}$ that reads x, replace x with new copy x_0. Create a new copy $a'_{t_2 i}$ and add x_0 into its parameter. Action $a'_{t_2 i}$ is triggered when Equation 1's first condition is triggered, Action $a_{t_2 i}$ is triggered when the second condition is triggered.
- New table t' has the new match fields, all actions from table t_2, and newly generated actions $a'_{t_2 i}$.

Memory usage. Memory usage varies depending on how many constants we can infer. Assume we can infer the value of x_0 and $\boldsymbol{\nu_1}$, then the newly generated table t' takes up $l_{t_2}(\sum_{i=0}^{n_{t_1 m}} w_{t_1 i} + \sum_{i=0}^{n_{t_2 m}} w_{t_2 i} + P_{t_2})$ memory. The newly generated table's match fields stays the same.

Multiple dependencies between two tables. Two tables can have more than one dependency and may not be limited to the same type. For example, they can have WAW and WAR dependency at the same time, or have two RAW dependencies. When dependencies have the same type, we can apply the pre-mentioned primitives directly (WAW) or recursively (RAW, WAR) to remove dependencies. For different dependency types, we choose not to remove them since the result table usually incurs too much memory overhead.

Counters, meters, and registers. In ASIC, stateful objects such as counters have two modes: direct and indirect. Direct counters have one-to-one mapping with table entries, while indirect ones have user-defined sizes. Depending on their mode and whether they are involved in the dependency directly (*i.e.* they write to variables read or written by another table), Cetus chose whether apply different primitives differently, and it is summarized in Table 1.

5.2 Table Merging Approach

Given a P4 program with n dependencies, there could be 2^n different table merging strategies at most. Different strategies produce different resource-usage trade-offs among stage, PHV, and memory. Rather than sending all of them to the constraint-based filter & optimizer module, we propose a heuristic algorithm that filters out strategies that violate basic constraints such as memory and stage.

In this approach, we only focus on comparing two metrics: stage saving and memory overhead. §6 would take more resources into account. If a strategy's memory overhead takes more stages than it can save by removing dependencies, it must end up occupying more stages than the original program, which conflicts with our goal. To sum up, given a P4 program, our heuristic algorithm runs as follows:

- Given a P4 program P, we generate its match-action DAG, D_P, and find all pairs of tables that potentially could be merged according to any of our primitives (mentioned in §5.1). Suppose we find n pairs.
- We build a binary decision tree T with n layers. Each layer represents one pair of tables, and each branch presents whether we remove the dependency of this pair of tables or not. Thus, a path from the root node of T to some leaf node of T represents a combination of table merging strategies.
- We thus run a deep-first search on T. During the searching process, we cut off the branches that violate basic memory and stage constraints. For each leaf node, we compute $S_{save} * m > M$, where S_{save} is the number of stages this strategy saves, m is the memory space of a single stage, and M is the memory overhead this strategy actually introduces. Note that S_{save} and M are computed by our primitives. If $S_{save} * m > M$, we keep this leaf node as one of our candidates used as the input of constraint-based filter & optimizer module (§6); otherwise, we drop this strategy.

6 Constraint-Based Filter & Optimizer

This module takes as input all candidates generated by §5, and then encodes each candidate program with all hardware resource size and constraints (stored in Cetus's backend DB) into an SMT formula. Then, we call an SMT solver (*e.g.*, Z3 [7]) to synthesize a table location plan that uses the least memory and stage resources. Finally, we realize this plan in a P4 program that specifies the locations of tables via `pragma` instructions.

The key challenge is how to efficiently solve these SMT formulas (each representing a candidate with all constraints). We found that the existing encoding approaches (*e.g.*, Lyra [10]) may result in state explosion, because a great number of diverse hardware resources create a huge search space that exceeds the SMT solver's searching capability.

To address the above challenge, we introduce a new approach that contributes two novel designs: (1) a new PHV encoding approach that significantly reduces the size of SMT formulas to avoid state explosion problem (§6.1); and (2) a two-step solving algorithm that decouples the solving process into table-related resource and variable-related resource solving to speed up the solving process (§6.2).

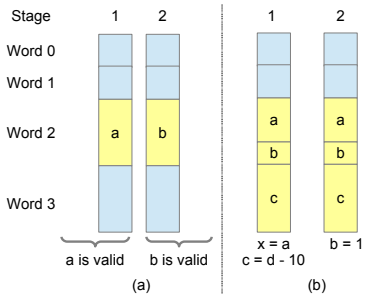


Figure 8: PHV sharing (a) across stages, (b) in one stage.

6.1 PHV Sharing Encoding

Packet Header Vector (PHV) serves as the bus between stages. The basic component of PHV is called word. There are tens of words with 8, 16, and 32-bit width respectively. One field can occupy one or multiple words. For example, a 48-bit source MAC field can take one 32b and one 16b word or three 16b words.

PHV is a scarce resource and needs careful planning, especially when the program is large and involves lots of headers and metadata. Simply adopting encoding approaches (*e.g.*, Lyra [10]) would waste the precious PHV spaces and fail to find a feasible solution. This is because Lyra's encoding assumes each word is dedicated to one variable; however, PHV words can be shared across variables in the PISA architecture, both across stages and within the same stage.

PHV sharing across stages. Different variables can occupy the same word at different stages. As shown in Figure 8(a), after stage 2, variable `a` is no longer used and another variable `b` can take over the same word. This allows us to use only one PHV container to store two independent variables that would otherwise require two containers. This sharing requires the variables have non-overlapping lifetimes, *i.e.* from the stage they are created till the last stage they are used. Note that all packet header fields' lifetime is the entire pipeline since they are created by the parser and consumed by the deparser. So the cross-stage sharing only applies to the metadata.

PHV sharing within one stage. Variables can also share the same word in the same stage as long as this sharing does not affect the correctness. Shown in Figure 8(b), variable `a` is read in stage 1 and variable `b` is assigned to a new value in stage 2. These two variables can share the same word. But variable `c` can not share with `a` at stage 1 because it was written by a subtract instruction. This is constrained by the fact that the Arithmetic Logic Unit (ALU) can perform at most one instruction to one word in one stage. The same-stage sharing applies to both header fields and metadata.

Cross-stage and same-stage sharing pack more variables into PHV, and it poses great pressure on PHV encoding. Because of the cross-stage sharing, we have to encode each stage's PHV allocation separately. The same-stage sharing further complicates the problem since we need to consider whether each pair of variables could share the same word.

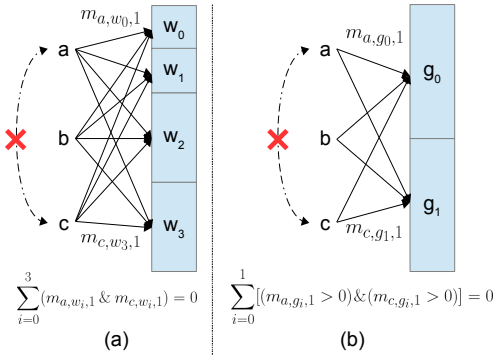


Figure 9: PHV encoding for 3 variables and 4 PHV words. (a) Strawman solution introduces 12 mapping variables and 4 rules. (b) Our solution reduces it to 6 mapping variables and 2 rules.

A strawman solution. A strawman solution is to encode the mapping $m_{v,w,s}$ between the variable v and each PHV word w at stage s. It encodes the cross-stage sharing by treating each stage separately. As for same-stage sharing, when two variables v_1 and v_2 cannot share the same word, we can add the constraint $m_{v_1,w,s}$ & $m_{v_2,w,s} = 0$. Next, we encode constraints such as each word has its own size limit, each variable should reserve enough bits in the PHV, *etc.* However, shown in Figure 9(a), because there are tens of stages and hundreds of PHV words, this solution introduces too many such mapping variables and the search space is huge.

Our encoding. Our PHV sharing encoding method addresses the scalability challenge. We observe that the total number of independent mappings in the encoded formula is the key complexity contributor. Thus, our focus is to reduce the independent mappings.

For same-stage sharing, we remove the boundary between PHV words and focus on whether variables can share with each other. We noticed that at each stage, there are only a few "shareable groups", the set of variables that can share with each other. Note that one variable can belong to multiple groups since the shareability is not transitive, *i.e.* variable v_1 can share with v_2 and v_3 cannot conclude v_2 can share with v_3. Then we can maintain the mapping between variables and these groups instead of the PHV words and restore PHV mapping afterward.

We also observe that in the encoded formula, all groups are symmetric: it does not affect the correctness when we reorder the groups. This is also another slow-down factor since it gives the SMT solver more freedom. To break the symmetry, we give preference to the groups with lower ID, the SMT solver can only use a new group until all the groups with lower ID are already assigned.

To summarize, the PHV encoding works as follows:

- (1) Given the input program $\mathcal{P}$, we count total number of non-assignment instructions I in each pipeline. This is the upper bound of the number of groups.

- (2) (Cross-stage sharing) For each variable v, we maintain: i) the mapping $m_{v,g,s}$, which denotes the number of bits v assign to group g at stage s, ii) the lifecycle l_v and r_v, which denotes the start and end stage of v.
- (4) (Same-stage sharing) If v_1 cannot share with v_2, then $(m_{v_1,g,s} > 0)$ & $(m_{v_2,g,s} > 0)$ is always false.
- (5) (Variable width) For each variable, if stage s is within the its lifecycle, the total number of bits in each group equals variable width b_v: $l_s \leq s \leq r_s \rightarrow \sum_i m_{v,g_i,s} = b_v$. Otherwise the summation is 0.
- (6) (PHV size) The summation of total number of bytes in each group should be less than PHV size. $\sum_i \lceil \sum_j m_{v_j,g_i,s}/8 \rceil \leq N_{PHV}$.
- (7) (Break symmetry) We prioritize groups with lower ID: $(\sum_j m_{v_j,g_{i+1},s}) > 0 \rightarrow (\sum_j m_{v_j,g_i,s}) > 0$.

In Figure 9(b), because only a cannot share with c, there are at most 2 shareable groups. We introduce 2 groups g_0 and g_1. Through this encoding, we can reduce the total mapping from 12 to 6. In reality, there is at least one order of magnitude fewer groups than the PHV words. This can greatly reduce the encoded formula's complexity.

6.2 Two-Step Solving

The PHV sharing encoding optimization can greatly reduce the encoded formula's complexity, but the SMT solver still struggles when dealing with large-scale production programs. Due to their scale, the encoded formula is still too complex. Additionally, PISA architecture's table-related resources (*i.e.* memory, table stage) and variable-related resources (*i.e.* PHV, crossbar) are orthogonal to each other: how much memory the table allocates per stage does not affect where the variable is located in the PHV. This loose coupling relationship forms a huge search space and exceeds SMT solver's searching capability under large scale programs.

While this loose coupling is the culprit, it offers us an optimization opportunity. We can safely ignore their correlation and split the SMT solving problem into two smaller problems. The two-step solving works as follows:

- Given a P4 program (*i.e.*, one of the candidates), we encode all table-related resources and constraints and find a feasible plan P_t meeting dependency and constraints.
- Upon P_t, we encode variable-related resources and constraints, and call the SMT solver to find a solution P_v capable of meeting resources (*e.g.*, PHV and crossbar) and constraints.
- If yes, with P_t and P_v, we have $P = P_t + P_v$ as a resource allocation plan for the input P4 program, returning plan P.
- If not, we return to step 1, find another feasible plan P'_t, P'_v.
- We repeat the above process until we find a valid plan P; otherwise, there is no valid plan for the input program.

This two-step approach can greatly improve the efficiency of our SMT solving. This aligns with our previous findings in §3.1 that the allocation of stages and table is our major concern. Other resources still remain and are more flexible.

6.3 The Best Result Selection

At the end of our workflow, the constraint-based filter & optimizer module may output one or more results that meet all already-known resource size and constraints. We select the most optimal one based on our internally-defined metric calculator. However, our experience shows that the constraint-based filter & optimizer module returns only one result in most cases.

7 Control Plane APIs Converter

After P' is obtained, our last task is to synthesize a control plane converter, making sure that the control plane APIs generated from the original program P are compatible with P' without any modification. Although different dependency removal primitives require different converting strategies, they follow the same underlying principle: generate new table entries that replace the previous tables' dependencies.

Due to limited space, we briefly describe the API converter for a concrete case shown in Figure 7(d) when installing new table entries. The rest of cases are detailed in Appendix A.

Let t_1, t_2 be the tables match c and e in program P, and t'_1, t'_2 be the tables after processing. In this example, t'_1 is the same as t_1. In the runtime, the converter keeps a record of existing entries in table t_1 and t_2 installed from the control plane.

When inserting an entry e_1 to table t_1, we first insert e_1 into table t'_1 unmodified. Next, for each existing entry e_{2i} in table t_2, create two new entries, one hits both e_{2i} and e_1, action is $b = p\_a$; one matches e_{2i} but misses e_1, action is $b = a\_0$. Insert all of them into table t'_2.

When inserting an entry e_2 to table t_2, for each existing entry e_{1i} in table t_1, we create two new entries as well. If table t_1 is empty, only create one rule that matches e_2 and other fields left wildcard. Other operations such as modifying or deleting an entry follow the same principle.

8 Deployment Experience

Cetus has been used to facilitate the development of P4 programs at Alibaba for one year. It has effectively decreased our P4 development workload by two orders of magnitude (from O(day) to O(min)) This section presents several real cases addressed by Cetus.

Case 1: Parallelizing network functions. A common problem our programmers frequently encountered is that implicit dependencies between actions or hardware constraints may prevent two or multiple network functions from occupying the same stages. If one of the functions contains a large table and another function consists of multiple small tables forming a long dependency chain, the total number of occupied stages could exceed the number of stages available, and our programmers had no clue on how to fix such a problem.

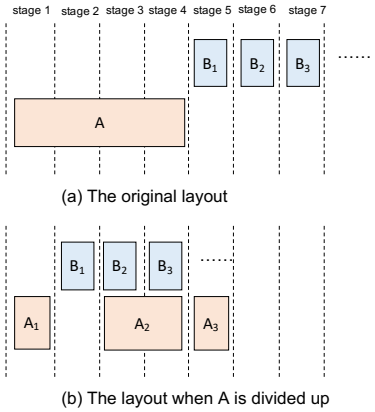


Figure 10: Parallelizing network functions via Cetus.

Figure 10 shows a real case in our edge gateway program. In the original P4 program, network function A only has one table A, which is a large table for load balancing. Function B consists of multiple tables like B_1, B_2, B_3, *etc.* formulating a chain of small tables, each of which being responsible for inserting customized metadata for diverse services. However, if the program places network functions A and B as shown in Figure 10(a), a fitting issue occurs because their resource usage exceeds total stages available. From the view of our programmers, they can only do trial and error.

Through the dependency removal algorithm introduced in Section 5, Cetus can automatically address this problem by parallelizing network functions within few minutes. As shown in Figure 10(b), Cetus detected there is a deep dependency between actions of A_1 and B_1, thus dividing function A into a few tables and maximizing the parallelization of table placement. We used the solution in [23] to guarantee the split tables act the same as the original one.

Case 2: Optimizing write-after-write dependency. Using global data is common in many programming languages and software systems. However, such practice comes with pitfalls in P4 programs. For instance, because the physical pipeline offers control registers, our programmers are allowed to explicitly drop a packet in packet validation, access control, and error handling. However, write operations to a common field issued by different modules may constitute write-after-write dependencies, which cause the number of required stages to exceed the actual stage number.

Figure 11 shows a real case. Figure 11(a) is the original P4 program. Two tables are invoked consecutively, which may call the same action to explicitly drop the packet. Because of write-after-write dependency, they must occupy two stages. Due to the "lengthy diameter" feature in our production programs, a fitting issue happened because stage resources are overly used. We therefore called Cetus to solve our fitting issue. Cetus automatically generates a program shown in Figure 11(b). We can observe that the two tables in the original program are merged into one, saving one stage to enable the program to compile. More interestingly, Cetus can also carefully merge the match keys from the two tables. Because the `color_drop` table does not match `meta.mirror` so the merged table used ternary to match `meta.mirror`.

```
action drop_packet(){
  eg_dprsr_md.drop_ctl = 1;
}

table color_drop(){
  key = { meta.pkt_color: exact; }
  actions = { drop_packet; NoAction; }
}

table mirror_drop(){
  key = { meta.pkt_color: exact;
          meta.mirror: exact }
  actions = { drop_packet; NoAction; }
}

control(){
  color_drop();
  mirror_drop();
}
```

(a) write-after-write dependency that requires two stages

```
action drop_packet(){
  eg_dprsr_md.drop_ctl = 1;
}

table color_mirror_drop(){
  key = { meta.pkt_color: exact;
          meta.mirror: ternary }
  actions = { drop_packet; NoAction; }
}

control(){
  color_mirror_drop();
}
```

(b) merged tables that require only one stage

Figure 11: Write-after-write optimization

```
action set_flow_tag(bit<16> tag){
  meta.tag = tag;
}

table color_flow(){
  key = { meta.ingress_port: exact; }
  actions = { set_flow_tag; NoAction; }
}

action set_sample_rate(bit<16> rate){
  meta.rate = rate;
}

table sample_rate(){
  key = { meta.tag: exact; }
  actions = { set_sample_rate;
              NoAction; }
}

control(){
  color_flow();
  sample_rate();
}
```

(a) read-after-write dependency that requires 2 stages

```
action set_tag_rate(bit<16> tag,
                    bit<16> rate){
  meta.tag = tag;
  meta.rate = rate;
}

table generated_tbl(){
  key = { meta.ingress_port: exact; }
  actions = { set_tag_rate; NoAction; }
}

control(){
  generated_tbl();
}
```

(b) merged tables that require only one stage

Figure 12: Read-after-write optimization

Case 3: Optimizing read-after-write dependency. Modularization is another common paradigm in program development. By clearly defining interfaces and decoupling modules, it allows the independent design and development of individual pieces of code. However, the modularization of P4 programs often comes at the expense of RAW dependencies.

In our production P4 programs, it is common for one module to set a particular field, which is later read by another module. Figure 12(a) shows a real program example where the table `color_flow` tags each packet depending on which port it comes from. Then, another `sample_rate` table sets the sampling rate based on a packet's tag. This constitutes read-after-write dependency; thus, `sample_rate` has to be placed at least one stage later than `color_flow`, resulting in at least two stages occupied. We found such read-after-write dependencies are quite annoying in our programs because many fitting issues were caused by this type of dependency.

With Cetus in hand, we directly applied Cetus in this scenario. Cetus automatically analyzes whether it is better to trade-off modularization for more efficient and compact code, given the limited number of physical stages in each pipeline. In particular, Cetus checks whether `meta.tag` is solely determined by `color_flow`, and whether they are applied consecutively. If so, it merges the two tables so that the first and second lookup are performed simultaneously within one stage, as shown in Figure 12(b). As a side effect, merging these two

Program	LoC	Table Num (Ig/Eg)	Before		After		Dependency Removed			Time
			Diameter (Ig/Eg)	Stage Num	Diameter (Ig/Eg)	Stage Num	WAW	RAW	WAR	
PINT [3]	380	13 / 0	6 / 0	7	6 / 0	6	0	2	0	19s
RTT [16]	408	12 / 0	9 / 0	9	8 / 0	8	0	3	0	25s
Bier [18]	703	26 / 4	7 / 2	11	5 / 2	7	2	8	2	41s
P4_protect [17]	576	12 / 1	5 / 1	6	4 / 1	4	0	6	0	25s
Conquest [5]	847	1 / 19	1 / 7	9	1 / 6	6	0	8	0	2m51s
Beaucoup [6]	1677	25 / 0	10 / 0	12	10 / 0	11	0	1	0	6m58s
P4_switch	4701	34 / 25	8 / 5	12	8 / 5	11	0	2	0	11m30s
CDN	6342	19 / 2	10 / 2	11	10 / 1	10	0	3	0	1m27s
Edge vSwitch	2733	32 / 6	9 / 3	11	8 / 2	8	2	3	0	1m21s
Edge Gateway	4417	32 / 37	8 / 7	12	8 / 7	11	2	1	1	7m21s

Table 2: Experimental results conducted on a workstation with Intel Xeon 2.5GHZ CPU and 128GiB RAM

tables may cause the new table to occupy more memory; however, as designed in §6, Cetus is able to take both factors (*i.e.*, memory and stage) into account and produces a feasible solution if such optimization is indeed worthwhile.

Case 4: SDE upgrade. As the programs keep evolving, we also upgrade the runtime and development-time infrastructure, including the versions of switch OS and the P4 compiler, to enjoy the latest performance optimizations and fixes provided by the vendors. In such an upgrading case, the program must be re-fit. We can consult Cetus to pinpoint the problem and search for a feasible table layout. After being automatically annotated with `pragmas`, the existing P4 program was successfully compiled while keeping its code structure intact. In this way, Cetus cleared the most challenging obstacle and enabled the upgrade of the whole system.

9 Evaluation

Our evaluation aims to answer whether Cetus can reduce different program's stage usage (§9.1) and how effective the optimization algorithms are (§9.2). All experiments were performed on a server with 2.5GHz CPU and 768GiB RAM.

9.1 Optimization

We chose 10 P4 programs, 6 open-sourced and 4 private ones, to evaluate whether Cetus can optimize and reduce their stage usage. In this evaluation, we mainly show Cetus's stage occupation reduction capability. For each program, we record its DAG's diameter and the number of stages it occupies in Υ chip before and after optimization. We further listed which types of dependencies Cetus removed and the time it took for each program. Table 2 shows the result.

First, Cetus removed 1 to 12 table dependencies, reduced the program's diameter by 1 to 2 and 1 to 4 stages. This shows the effectiveness of the primitives used by Cetus and our findings in §3.1 also apply to open source programs.

Second, Cetus can successfully find the best candidate at a decent speed. For simple programs, Cetus can find a plan in under a minute. For complicated ones, Cetus still managed to finish the search in minutes. Compared with the days of efforts developers spent optimizing the program manually, this is way faster and saves a lot of deployment efforts.

Third, we can see that most of the dependencies removed were RAW dependencies. This is because of two reasons: (1) RAW dependency is common in programs. (2) RAW dependency is hard to find and also hard to remove. For example, below is a code snippet from Beaucoup [6]:

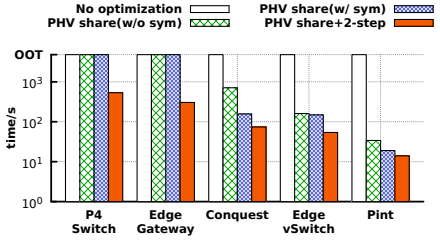


Figure 13: Time for a solution under different optimizations.

```
if(ig_md.cf_key_matched==1) {
    exec_regcoupon_merge(); // writes coupon_merge_check
}
if (ig_md.cf_decay_has_expired==1) {
    exec_counter_set_to_one();
} else {
    if (ig_md.cf_key_matched==1 && ig_md.coupon_merge_check==0) {
        exec_counter_incr();
}}
```

In the above code, the action `exec_regcoupon_merge()` writes variable `coupon_merge_check`, which is later read by the condition of action `exec_counter_incr()`. Cetus removes their dependency through the RAW dependency removal primitive, and it reduces one stage occupation. But for developers, it is hard to notice because it is spread across two different condition branches far away.

9.2 Performance

To further evaluate the effectiveness of the optimization techniques introduced in §6, we chose several typical programs with different scales and run experiments with different optimization techniques enabled. Starting from the naive solution with no optimization, we add vanilla PHV sharing encoding, symmetry breaking encoding, and finally two-step solving to Cetus sequentially. We set 1 hour as the timeout threshold. The result is shown in Figure 13.

Without any optimization, all programs timed out, which means it is necessary to introduce optimizations. For small-scaled programs, such as Conquest, Edge vSwitch, and Pint, adopting PHV sharing encoding can greatly improve the performance, indicates that the bottleneck lines in the complexity of the encoded SMT formula. However, for large-scale pro-

grams, such as P4 Switch and Edge Gateway, we only met the deadline after adding all three optimizations. This shows that for large scale programs, encoding optimization is not enough, the search space is still too large for the SMT solver to handle. It is necessary to leverage the key findings in §3.1 and bring in two-step solving to give a hint to the SMT solver.

10 Discussion and Lessons

This section discusses our lessons and limitations.

Is P' functionally identical to P? In principle, Cetus's approaches, including table merging and constraint-based filter & optimizer, can only change and optimize the location of tables, rather than the function logic of programs; thus, P' should be functionally the same as P. While we have not manually proved our approach on this property, in Alibaba, we employ a P4 verification tool, Aquila [27], to check the consistency between P and P' when Cetus generates P'. If Aquila returns "yes", that means we can use P' to replace P. So far, we have not seen any inconsistency case.

Can Cetus capture all hardware constraints within Υ chip? We encode constraints as many as we can; thus, we can only make sure that P' will not violate any constraints we have encountered before. With the accumulation of more and more hardware constraints, we believe the capability of Cetus will become stronger. However, we cannot guarantee every P' can compile to Υ chip. We did experience few cases that P' does not compile due to unknown constraints.

Can lengthy diameter always hold? We cannot guarantee the lengthy diameter can always exist in our production programs in the future; however, based on our experience with Cetus so far, the stage shortage issue resulting from the lengthy diameter is still the highest priority barrier in our scenario. We thus suggest the ASIC vendor consider releasing a chip with double the number of stages and less memory.

Cetus's limitations. We have the following main limitations. First, Cetus can only remove dependencies like WAW, RAW, and WAR. Cetus cannot handle more tricky cases such as removing dependency via modifying program semantic. Both RAW dependency removal algorithms require a third table in front to parallelize the latter two tables. For programs such as Syncookies [22], Cheetah [29], because they have long, chained sequential computations, the requirement of RAW dependency removal is not met, Cetus cannot perform optimizations. Second, Cetus cannot optimize a program when it occupies too many resources, since the dependency removal algorithms come at the cost of additional resources in the switch, such as PHV and memory. Third, we cannot guarantee Cetus's implementation is bug-free although we spent a lot of time checking our implementation bugs; thus, sometimes the output P' may not be the best one. Finally, if a new programmable ASIC architecture is introduced, Cetus cannot be directly used to generate compilable programs for this new ASIC. Cetus has to encode all constraints of this new ASIC.

11 Related Work

P4 program optimizers and compilers. This type of systems optimize resource usage in programmable ASICs or simplify programmers' tasks on expressing their coding intent. P4All [13, 14] aims to optimize resource usage by leveraging reusable data structures, such as bloom filters and key-value stores; however, our production P4 programs do not share these data structures. P4visor [31, 32] optimizes resources by merging redundant code fragments (*e.g.*, header parser and tables). P4visor is a good complementary to Cetus. Before Cetus was developed, we already built an internal system (similar to P4visor) to merge redundant code fragment. μP4 [26] proposes a modular way to write P4 code. Jose *et al.* [15] compiles P4 programs to architectures such as the RMT and FlexPipe. Domino [24] and Lyra [10] simplify data plane programming by specifying C-like new languages. Chipmunk [11, 12] leverages slicing, a domain-specific synthesis technique, to remove unnecessary resources cost by Domino. P^{2}GO [30] proposes an idea that reduces the allocated resources of a P4 program based on traffic trace profiling. However, it might be hard for us to deploy it in our environment, because if unexpected traffic turns up after the profiling, some function might be already pruned. Different from the state of the art (that keeps the original dependencies), Cetus optimizes resource usage by removing dependencies in P4 programs.

Network-wide configuration synthesis. Configuration synthesis work [4,8,9,19,21,28] offers the operator network-wide abstractions for configuration synthesis. SyNET [8] and ConfigAssure [19] offer general abstractions to synthesize the protocol configuration. Recent work [9]indicates that none of the above systems is scalable to cloud-scale networks. Propane [1, 2], Snowcap [21], and Jinjing [28] synthesize BGP, updating, and ACL configurations, respectively.

12 Conclusion

We have presented Cetus, the first system that releases the P4 programmers from frustrating trial and error compiling. Cetus can automatically convert an uncompilable P4 program into a functionally identical but compilable P4 program. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from O(day) to O(min)).

This work does not raise any ethical issues.

Acknowledgments

We thank our shepherd, Dejan Kostic, and NSDI'22 reviewers for their insightful comments. We also thank Vladimir Gurevich for his valuable feedback on both the technical part and the presentation of this paper. This work is supported by Alibaba Group through Alibaba Research Intern Program. Yifan Li is supported in part by the National Natural Science Foundation of China under Grant Number 61872212.

References

[1] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[2] Ryan Beckett, Ratul Mahajan, Todd D. Milstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 662–680, 2020.

[4] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[5] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 15–29, 2019.

[6] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 226–239, 2020.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[8] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)*, 2017.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. NetComplete: Practical network-wide configuration synthesis with autocmpleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[10] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 435–450, 2020.

[11] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019.

[12] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 44–61, 2020.

[13] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[14] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[16] Elie Kfoury, Jorge Crichigno, Elias Bou-Harb, and Gautam Srivastava. Dynamic router's buffer sizing using passive measurements and p4 programmable switches.

[17] Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. P4-protect: 1+ 1 path protection for p4. In *Proceedings of the 3rd P4 Workshop in Europe*, pages 21–27, 2020.

[18] Daniel Merling, Steffen Lindner, and Michael Menth. Hardware-based evaluation of scalable and resilient multicast with bier in p4. *IEEE Access*, 9:34500–34514, 2021.

[19] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16(3):235–258, 2008.

[20] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 194–206, 2021.

[21] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 33–49, 2021.

[22] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. Me love (syn-) cookies: Syn flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221*, 2020.

[23] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1):36–47, 2001.

[24] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[25] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 731–747, 2021.

[26] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 329–343, 2020.

[27] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 17–32, 2021.

[28] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the 2019 ACM SIGCOMM Conference*, pages 214–226. 2019.

[29] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *The 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[31] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[32] Peng Zheng, Theophilus A. Benson, and Chengchen Hu. Building and testing modular programs for programmable data planes. *IEEE J. Sel. Areas Commun.*, 38(7):1432–1447, 2020.

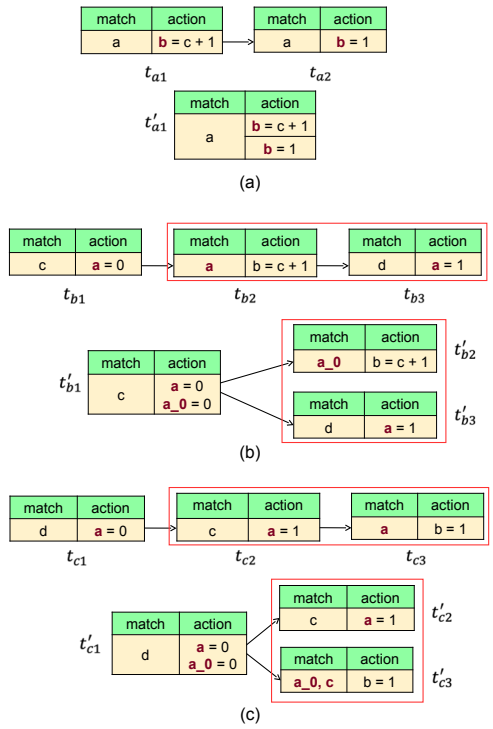


Figure 14: Examples for control plane APIs converter: (a) WAW (b) WAR (c) RAW-match.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A Control Plane APIs Converter

This section details how Cetus's control plane API converter bridges the inconsistency between the original program P and the optimized one P'. We labeled the tables in Figure 7 and show the example tables in Figure 14.

Write-after-write dependency. Since two tables t_{a1} and t_{a2} in Figure 14(a) share the same match field, the entries for both tables are inserted to the merged table t'_a directly. However, when two entries e_1, e_2 for t_{a1} and t_{a2} respectively overlaps their match field (*e.g.* e_1 matches 10.0.0.0/8 while e_2 matches 10.0.0.0/16), entry e_2 has higher priority than e_1 because table t_{a2} applies later than t_{a1}.

Write-after-read dependency. The match field of table t_{b2} is renamed. For an entry e_2 inserted to table t_{b2}, Cetus renames the match fields' name and inserts it to table t'_{b2}. For example, in Figure 14(b), the match field a in e_2 is renamed to a_0. Entries for table t_{b3} are inserted to table t'_{b3} directly.

Read-after-write-match dependency. In this case, Cetus records all the entries inserted to table t_{c2} and t_{c3} in a 'logical table' stored in memory. When a control plane application inserts an entry e_2 to table t_{c2} with match value c_{e_2}, Cetus first inserts e_2 to table t'_{c2} unmodified. Next, if there exists an entry recorded in logical table t_{c3} that matches the result of action in table t_{c2}, which is $a = 1$ in Figure 14(c), then Cetus creates a new entry e'_2 that matches c with value c_{e_2} and ignores value of a_0 and inserts it to table t'_{c3}. When an entry e_3 is inserted to table t_{c3}, there are two cases. If e_3 matches the result of the action in table t_{c2}, record it in the 'logical table' and do not insert it anywhere. Otherwise, rename the match field name of e_3 from a to a_0, add another match field c in e_3 but ignores the value. The 'ignore' can be expressed by using the wildcard if t'_{c3} uses TCAM memory, or by enumerating all possible values if it uses SRAM memory.

Read-after-write-action dependency. This part has been detailed in §7.

The entry removal operation is the reverse of the above actions.