Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Traffic-Aware Compilation

Semester Thesis

Author: Patrick Wintermeyer

Tutors: Maria Apostolaki, Alexander Dietmüller

Supervisor: Prof. Dr. Laurent Vanbever

March 2020 to June 2020

Disclaimer

To the best of my knowledge and in accordance with the Declaration of Originality, the following specifies the main writing contributors to each Chapter in order of importance:

1 Abstract: Maria Apostolaki

2 Chapter 1: Maria Apostolaki, Laurent Vanbever, Patrick Wintermeyer

3 Chapter 2: Alexander Dietmüller, Patrick Wintermeyer, Maria Apostolaki

4 Chapter 3: Alexander Dietmüller, Maria Apostolaki

5 Chapter 4: Patrick Wintermeyer, Maria Apostolaki

6 Chapter 5: Patrick Wintermeyer, Maria Apostolaki

7 Chapter 6: Maria Apostolaki

8 Chapter 7: Patrick Wintermeyer

Abstract

Programmable devices allow the operator to specify the data-plane behavior of a network device in a high-level language such as P4. The compiler then maps the P4 program to the hardware after applying a set of optimizations to minimize resource utilization. Yet, the lack of context restricts the compiler to conservatively account for all possible inputs – including unrealistic or infrequent ones – leading to sub-optimal use of the resources or even compilation failures. To address this inefficiency, we propose that the compiler leverages insights from actual traffic traces, effectively unlocking a broader spectrum of possible optimizations.

We present a system working alongside the compiler that uses traffic-awareness to reduce the allocated resources of a P4 program by: *(i)* removing dependencies that do not manifest; *(ii)* adjusting table and register sizes to reduce the pipeline length; and *(iii)* offloading parts of the program that are rarely used to the controller. Our prototype implementation on the Tofino switch automatically profiles the P4 program, detects opportunities and performs optimizations to improve the pipeline efficiency.

Our work showcases the potential benefit of applying profiling techniques used to compile general-purpose languages to compiling P4 programs.

Contents

Chapter 1

Introduction

1.1 Motivation

Thanks to programmable data planes, network programmers can now define the forwarding behavior of their switches using programming languages such as P4 [13]. To ensure portability across platforms, these languages abstract away many hardware details and rely on a compiler to "map" programs to the available hardware resources. Typical hardware resources include the number of processing stages, the amount of memory available in each stage, as well as the number of operations available per packet. Compiling a P4 program so that "it fits" the particularly tight resource budget of typical switches [23] is a challenging problem that existing P4 compilers approach with various optimizations techniques [17].

While useful, existing compiler optimizations are also inherently limited in that they cannot reason about runtime information, as they only have access to the source code. Among others, this forces them to conservatively account for all possible inputs – including unrealistic and infrequent ones – leading to sub-optimal use of the resources or even compilation failures. In particular, the network traffic can be such that some parts of a P4 program might be seldom executed while occupying a significant amount of the allocated resources. Clearly, knowing such an execution profile would enable further compilation optimizations, *e.g.,* allowing to save stages or to reduce memory consumption. Such profile-guided optimizations are well-known in general-purpose programming languages and are available in many production-grade compilers (*e.g.,* PGO in Clang [4], BOLT [22], Propeller [8]).

1.2 Profile-guided optimization on programmable data planes

We argue that profile-guided optimization should also be applied to programmable data planes and introduce **P^{2}GO**.

Using runtime information, P^{2}GO automatically optimizes a P4 program so that it requires less resources. More specifically, given a (representative) packet trace and a set of forwarding rules, P^{2}GO profiles the P4 program by observing the execution path taken by each packet. P^{2}GO then uses this profiling information to adapt the P4 program such that it uses strictly less hardware resources after compilation.

Despite being intuitive, we believe profile-guided optimizations open up a rich research agenda for our community. This agenda includes research questions such as: *How to compute representative execution profiles? Which optimization techniques bring the most benefits? How do we optimize multiple resources simultaneously? Should we allow possibly unsafe optimizations that may change*

the program's semantics? How do we deal with changes in the profile?

We start to answer these questions by optimizing "only" one resource, albeit a fundamental and often limiting one: the number of pipeline stages. We introduce three profile-guided optimizations to reduce the number of stages, all of which go beyond the capabilities of available P4 compilers. In particular, we first show that profiling can help uncover "fake dependencies", *i.e.,* dependencies that are reported by static analysis but do not manifest in practice. We then show that profiling can shed light on opportunities for memory optimizations by ensuring that they do not change the overall behavior of the P4 program. We finally show that profiling can uncover code segments that are barely used but consume a significant amount of resources. Such segments are good candidates to be offloaded to software.

P^{2}GO preserves the semantics of the original program when executed on the original trace. Concretely, if the traffic trace is not representative, then the behavior of the program might diverge from the original one. P^{2}GO addresses this problem by directly involving the programmer in the optimization process. Specifically, P^{2}GO reports to the programmer the various adaptations it made to the original program, together with the profile-based observations that allow them. The programmer can then choose to selectively accept them or not based on her knowledge of the general traffic. Exposing only those profile-based observations that guided the performed optimization allows the programmer to worry less about how representative the initial traffic trace is.

1.3 Novelty

Existing P4 compilers assume that the P4 program is static and needs to be mapped to hardware resources without any modifications. P5 [11] challenged this assumption by modifying the P4 program such that it leads to a more efficient allocation. In contrast to P^{2}GO, P5 is *not* profile-guided and instead requires high-level policies as input to the optimization stage. P5 is, therefore, only applicable to use cases where such high-level policies exist. P5 is also limited in that it can only adapt the P4 programs in a coarse-grained manner, deactivating entire code blocks. In contrast, thanks to profiling, P^{2}GO can *discover* high-level policies and implementation-level inefficiencies.

1.4 Task and goals

The task is to design a process P^{2}GO that optimizes the number of pipeline stages through three optimization techniques, while leveraging insights gained from its traffic profile:

1 Remove dependencies that do not manifest in the traffic profile

2 Optimize resource usage by resizing memory allocations

3 Offload rarely used code segments to other devices or to the controller

Furthermore, this process is to be illustrated and evaluated through a proof-of-concept implementation featuring the most important components of P^{2}GO.

The goal is to show the potential benefits of applying traffic profiling techniques to compiling P4 programs and open new research directions for future work.

1.5 Overview

Chapter 2 gives background information on constraints imposed by the architecture on the mapping of a P4 program to the resources of a programmable switch and presents related works.
Chapter 3 describes P^{2}GO's *modus operandi* through the illustration of a concrete example and how P^{2}GO profiles and optimizes it.
Chapter 4 describes the design of P^{2}GO and its proof-of-concept (PoC) implementation, while Chapter 5 features a preliminary evaluation carried out using the PoC.
Chapter 6 outlines future work and research directions.
Chapter 7 compactly summarizes our work.

Chapter 2

Background and Related Work

2.1 Background

The following section will present the necessary background information necessary to understand P^{2}GO's optimizations and when such opportunities arise.

2.1.1 P4 Language

P4 [13] is a language to program packet processing pipelines independently of specific hardware targets. On a high level, a P4 pipeline consists of a programmable parser that translates incoming bytes into packet headers defined by the programmer, followed by a series of match-action tables that modify these headers. The 'control flow' of the P4 program defines the conditions for each table to be applied, while each table defines which header fields are matched (and how), and which actions may be executed. Actions may modify header fields and read or write stateful memory such as counters, meters, and registers. Importantly, the P4 program only defines which types of matches and actions are possible for each table. The actual mapping from particular header values to concrete actions is *not* known at compile time, but only specified at runtime by installing *match-action rules*.

2.1.2 Architectural Constraints

P4 programs can be compiled to a wide range of software and hardware targets, such as software switches [3], eBPF [5], FPGA NICs, or ASIC switches with programmable data planes. In particular hardware targets cannot support arbitrary packet processing pipelines. Typically, hardware targets implement a multi-stage pipeline with memory allocated to each stage in the pipeline, i.e. the memory is usually not shared between stages. Computational resources are similarly bound to stages. Consequently, typical hardware constraints include the total number of available stages (i.e. the longest possible path), and the amount of memory as well as computational resources per individual stage.

Target-specific P4 compiler backends must optimize the mapping of logical tables to physical stages to ensure that the program fits onto the hardware target. If the resulting mapping requires too many physical stages, the program cannot be compiled to the target. Specifically, if a table might require more resources than available in a single stage, the compiler must distribute the table across multiple stages. On the other hand, tables which do not require many resources, can be applied concurrently in the same stage, which allows packing them more densely on hardware. Independent tables even allow reordering to use resources more efficiently. This is impossible if

tables depend on another, in which case they have to be placed in different stages.
Important dependencies between two tables `A` and `B`, or between a table `A` and a conditional control statement `C` are:

- *Match Dependency:* An action of `A` modifies a header field that either `B` matches on, or which is used in `C`. `B`/`C` can only be applied/evaluated in a stage after `A`.
- *Action Dependency:* An action of `A` modifies a header field that an action of `B` reads or modifies. `B` can only be applied in a stage after `A`.
- *Reverse Read:* An action of `A` reads a field that `B` modifies.
- *Table Predication:* Whether `B` matches depends on whether `A` has matched.
- *Control Flow:* Independent.
- *Conditional Execution Dependency:* Whether `A` is applied depends on the result of `C`. `A` can only be placed in a stage where `C` can be evaluated.

From the above, it is clear that dependency optimization opportunities arise especially between *match-* and *action-dependencies*, as they specifically do not allow placement of tables in the same stage.

2.2 Related Work

In this section we present previous work associated with P^{2}GO and group them into two categories that P^{2}GO combined: profiling execution paths and optimizing code for programmable switches.

2.2.1 Profiling execution paths

Previous work has explored tracking packets in the data plane's control logic to provide visibility to the programmer. P4DB [25] uses match-action tables and packet digests to report a packet's path to a debugging platform. Another approach [20] has explored online tracking of packets in the data plane through a variation of the Ball-Larus encoding, which tracks all possible execution paths. In this case, per-packet information is stored in metadata during execution. While simpler, P^{2}GO's offline profiling approach is adequate to guide our optimizations. Future work could pair powerful online profiling with P^{2}GO's optimization phases to allow real-time optimizations (see §6.1).

2.2.2 Optimizing code for programmable switches

There exist numerous approaches to optimizing the placement of SDN policies through virtualization and network abstraction [18, 19, 24]. These works optimize rules needed to enforce endpoint and routing policies by using a global view of the network. Similarly, [21] focuses on rules, applied to both hypervisors and switches. SNAP [12] implemented a new high-level programming language to efficiently distribute state, processing, and routing amongst devices of a network, by translating SNAP code to low-level switch code (NetASM). Unlike previous works, P^{2}GO operates on a per-device level and on the RMT architecture [14]. Particularly, P^{2}GO directly optimizes P4 code, which is more expressive. Also, special considerations need to be taken for stateful memory and metadata, on top of the actual packet processing, which is entirely customizable. P5 [11] also optimizes P4 code by removing dependencies between high-level features if they do not manifest in the high-level provided by the programmer. On the contrary, P^{2}GO relies on information produced by profiling, which is more fine-grained. Moreover, P^{2}GO achieves more general optimizations. Concretely, P^{2}GO optimizes dependencies on the level of individual actions and tables, opposed to features, and also considers target-specific memory optimizations and offloading code segments.

Chapter 3

Modus Operandi

In this section we walk through the compilation of an example P4 code 1. We first describe the functionality of the P4 code §3.1, before we explain how the compiler maps it to hardware §3.2.

Example 1 P4 program with multiple functions. The percentages show the *hit rate* for each table determined by profiling.

```
1:  control                                                          hit rate
2:      if valid(ipv4) then
3:          apply(IPv4)                                              100%
4:          if valid(udp) then
5:              apply(ACL_UDP)                                       8%
6:              if valid(dhcp) then
7:                  apply(ACL_DHCP)                                  14%
8:              if valid(dns) then
9:                  apply(Sketch_1)                                  2%
10:                 apply(Sketch_2)                                  2%
11:                 apply(Sketch_Min)                                2%
12:                 if sketch_count >= 128 then
13:                     apply(DNS_Drop)                              1%
14: end control
```

3.1 Example P4 program

A network operator has programmed a device to support IP forwarding, simple UDP-based access control list (ACL), DHCP and DNS server protection [9] (Ex. 1). First, the program takes care of IPv4 forwarding (Table `IPv4`): It sets the egress port or drops packets with unknown destinations. Next, the network operator applies two ACL tables: one filtering UDP packets in general and one targeting DHCP packets in particular. The former (table `ACL_UDP`) can drop packets for specific UDP ports, while the latter (table `ACL_DHCP`) can drop DHCP packets based on the offered IP address of DHCP replies. Finally, the program keeps track of the number of packets per DNS flow using a Count-Min Sketch (CMS)[16] with two hashes. The hash functions require two separate registers (the tables `Sketch_1` and `Sketch_2` match on the respective hash values and query/update the registers). The sketch output (`sketch_count`, stored in packet metadata) is the minimum of

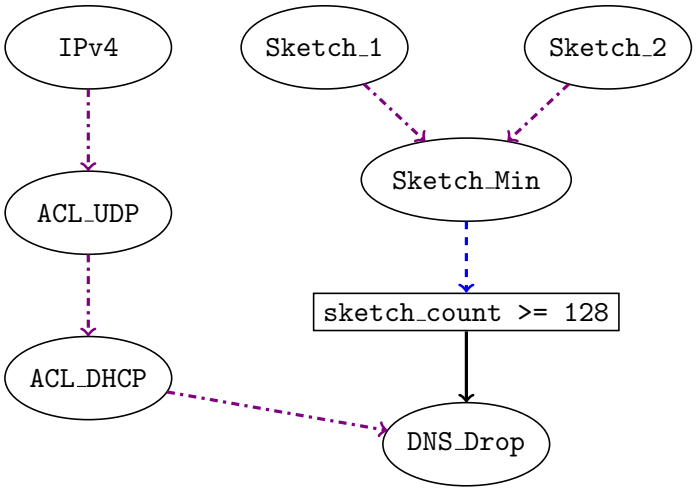

Figure 3.1: Dependency graph for the example program (Alg. 1). Round nodes are tables, square nodes are conditional control statements. Solid black arrows are *conditional execution dependencies*, dashed blue arrows are *match dependencies*, and dash dotted violet lines are *action dependencies*.

all hashes (table `Sketch_Min`) If the output exceeds a threshold, packets from the flow are dropped (table `DNS_Drop`).

3.2 Compiling the P4 program

Next, the programmer submits her P4 code for compilation to a target-specific P4 compiler. The compiler is responsible for mapping the P4 tables to the hardware target, namely an ingress and an egress pipeline each composed of a series of fixed-order stages. We will first describe the compilation process and then the output of the compiler.

3.2.1 Mapping the program to hardware

In doing so, the compiler tries to minimize the stages used to make the pipeline more efficient. Two main factors prevent the compiler from putting tables in the same stage: *(i)* dependencies among tables; and *(ii)* the limited amount of memory available to each stage. First, two tables are dependent if the execution of one depends –and thus must succeed– the other's execution. In our example `DNS_Drop` will only be executed under a condition (packet's flow has sent more than 128 packets already), which can only be evaluated after the execution of `Sketch_Min`. Thus, tables `DNS_Drop` and `Sketch_Min` need to be put in different stages. Second, each stage contains a limited amount of memory that can be used by the tables mapped in the stage[1]. In our example `Sketch_1` and `Sketch_2` use stateful memory, namely a register array. Yet, the sum of the size of the arrays they use exceeds the memory of a single stage. Thus, `Sketch_1` and `Sketch_2`, need to be put in different stages.

3.2.2 Compiler output

The compiler produces the binary to run in the target. Moreover the programmer can retrieve: *(i)* the final allocation seen in Table 3.2 *(ii)* the dependency graph containing all dependencies among

[1]this is highly simplified due to NDA

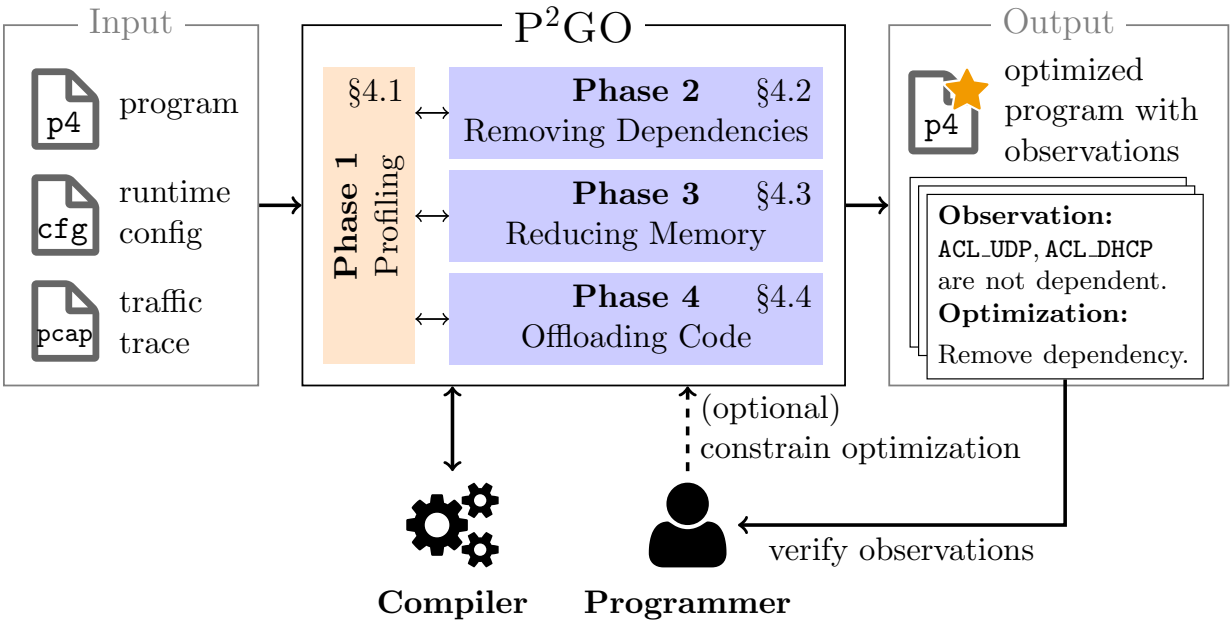

Figure 3.2: P²GO works alongside the compiler, performs a series of profile-guided optimizations to produce an optimized program which requires less resources. P²GO returns the profile-based observations which guided the optimizations and which the operator needs to verify.

tables seen in Fig. 3.1; and *(iii)* the control graph containing all possible execution paths packets may take throughout the program.

3.3 Profile-guided optimization with P²GO

P²GO aims to optimize P4 programs by leveraging profiling techniques (Fig. 3.2). Aside from the program, the programmer needs to provide two inputs to bootstrap P²GO: the *initial* runtime configuration of the program (*i.e.,* the match-action rules installed in the tables) and a trace of incoming traffic. For individual devices, these inputs can be recorded with relative ease assuming that the network programmer has access to the device of interest.[2] P²GO works *alongside* the compiler; it iteratively modifies and re-compiles the P4 program, and analyzes the compiler output. P²GO returns an optimized P4 program – which has the same behavior as the original program for the given traffic trace – together with a summary of profile-based observations that guided each modification of the original program. For example, P²GO might return the observation that the memory of a table can be reduced without impacting the program's behavior. If the programmer agrees with these observations, she can use the optimized program or even re-submit it to P²GO to optimize it further. Otherwise, she can re-run P²GO with one or more optimizations disabled.

P²GO operates in four phases. In the first phase, it *profiles* the behavior of the program. In the following three phases, it *optimizes* the program guided by the profile.

3.3.1 Phase 1: Profiling

Using the traffic trace and runtime configuration, P²GO observes the execution path of each packet to create the program *profile*. The profile provides two main insights: *(i)* the *hit rate* of each table,

[2]In practice, some match-action rules may be added by a central controller managing multiple interdependent devices. This case requires network-die optimization. We discuss this in §6

Sets of non-exclusive actions
{IPv4, ACL_UDP}
{IPv4, ACL_DHCP}
{IPv4, Sketch_1, Sketch_2, Sketch_Min}
{IPv4, Sketch_1, Sketch_2, Sketch_Min, DNS_Drop}

Table 3.1: During profiling, P^{2}GO observes whether sets of actions are *non-exclusive*, i.e., are applied to the same packet(s). Each item represents a *specific action* of the table; the action names are omitted for brevity.

which is the percentage of packets the table matched; and *(ii)* the sets of non-exclusive actions that are applied to the same packet(s). The annotation of Ex. 1 and Table 3.1 give an example profile for our program.

3.3.2 Phase 2: Removing dependencies

In this phase, P^{2}GO compares the dependency graph with the profile to detect dependencies that do not manifest in practice. P^{2}GO observes that while `ACL_DHCP` depends on `ACL_UDP` (see Fig. 3.1), there is no set of non-exclusive actions that contains the dependent actions of both tables (see Table 3.1). In other words, these actions are never applied to the same packet. P^{2}GO modifies the program to only apply `ACL_DHCP` if `ACL_UDP` misses, which allows the compiler to ignore their dependency. For the modified program, the compiler places both tables in the same stage, shortening the pipeline by a stage (see Table 3.2). Note that P5 would not be able to remove such a dependency as an operator might need both ACLs.

3.3.3 Phase 3: Reducing memory

In this phase, P^{2}GO searches for opportunities to shorten the pipeline without changing the program's behavior by slightly reducing the allocated memory. In our example, P^{2}GO first finds that reducing either the memory allocation for table `Sketch_1` or table `IPv4` also reduces the number of required stages (see §4.3). Starting with `Sketch_1`, as it has a lower hit rate than `IPv4`, P^{2}GO profiles the reduced-memory program with the same runtime configuration and traffic trace and discovers that the program behavior is changed: The hit rate of `DNS_Drop` has increased. (The memory reduction increases the false positive rate, causing DNS queries to be overcounted.) Thus, P^{2}GO discards this optimization of `Sketch_1`. Following, P^{2}GO considers `IPv4`, finds that reducing its memory does not change the program behavior and applies this modification. The optimized program occupies one stage less (see Table 3.2).

3.3.4 Phase 4: Offloading code to controller

In this phase, P^{2}GO examines if parts of the program could be offloaded to the controller. P^{2}GO observes that DNS branch of Ex. 1 has low hit rates in the profile (2%), yet utilizes a significant amount of resources. Thus, offloading the `Sketch*` and `DNS_Drop` tables to the controller could shorten the pipeline without overloading the controller (as the tables are rarely matched). To do so, P^{2}GO replaces the whole branch with a table that forwards DNS packets to the controller. As

Stage	1	2	3	4	5	6	7
Initial Program	IP	IP	AU	AD S1	S2	SM	DD
Removing Deps.	IP	IP	AU AD S1	S2	SM	DD	-
Reducing Memory	IP	AU AD S1	S2	SM	DD	-	-
Offloading Code	IP	AU AD C	-	-	-	-	-

Table 3.2: After each optimization phase, the program from Ex. 1 requires less stages. Each box represents stage memory allocated to a table: IP `IPv4`, AU `ACL_UDP`, AD `ACL_DHCP`, S1 `Sketch_1`, S2 `Sketch_2`, SM `Sketch_Min`, DD `DNS_Drop`, C `To_Ctl`.

a result, the pipeline requires three stages less Table 3.2.[3]

P^{2}GO reserves code offloading as the last phase to allow optimizing the data plane first. As an intuition, if this was the first phase, P^{2}GO might have offloaded both ACLs, originally requiring two stages. Yet after removing dependencies, the tables require only one stage, and offloading them has no benefits. Note that P5 would not remove this segment as it is used.

[3]Observe that removing any single one of those tables alone would not decrease the traffic moved to the controller.

Chapter 4

Design

Using the Tofino simulator and compiler (SDE 8.2) provided by Barefoot, P^{2}GO optimizes programs written in $P4_{14}$.[1] We have implemented a PoC of P^{2}GO in python using ∼300 LoC for profiling and ∼1200 LoC for the optimization phases.

In the following, we elaborate on each phase of P^{2}GO from a conceptual point of view.

4.1 Profiling P4 programs

The goal of profiling is to trace the execution paths that packets take through the P4 program. To that end, P^{2}GO first modifies the program such that each packet is marked with the sequence of actions applied to it. Next, P^{2}GO replays the traffic trace as input to the modified program and collects the outgoing packets. From the marked packets, P^{2}GO can infer which actions/tables have been executed and which were applied on the same packet.Next, we elaborate on how P^{2}GO instruments the program and creates the profile.

4.1.1 Instrumenting the program

P^{2}GO modifies the program to append a profiling header after the original headers of each packet. The profiling header contains multiple fields, each corresponding to an action. Each field is set when the corresponding action is executed. Note that these modifications have no impact on the behavior of the actual program, as the instrumented program is only used during profiling.

4.1.2 Building the profile

P^{2}GO loads the modified program in the Tofino simulator, installs the provided match-action rules, and replays the traffic trace while collecting outgoing packets with profiling headers. By processing the collected packets, P^{2}GO infers a *profile* consisting of: *(i)* the fraction of packets that match each table (hit rate); and *(ii)* the sets of actions that are applied to the same packet(s) (non-exclusive actions).

4.2 Removing Dependencies

The goal of this phase is to find tables that are seemingly (*i.e.,* according to static analysis) dependent but practically mutually exclusive (*i.e.,* according to profiling) and to remove their dependency.

[1] We refrained from using bmv2 [3] as our evaluation requires realistic resource allocation.

In this case, P^{2}GO modifies the program to explicitly express that the two tables can fit in the same stage. Such opportunities are common: to make programs more reusable, programmers often write the control pipelines of their programs as a sequence of apply statements, even though each packet only matches a subset of them. In principle, it is possible for the programmer to use special annotation (*e.g.,* pragmas) or write the p4 control flow such that it explicitly expresses that certain pairs of tables are mutually exclusive. In practice, though, considering all possible dependency pairs can quickly overwhelm the programmer.

4.2.1 Identifying unseen, yet restrictive dependencies

P^{2}GO considers as candidates for removal only dependencies (1.1) that are in the longest path of the dependency graph, as only those have the potential to reduce the pipeline length. Among the candidates, P^{2}GO removes a dependency if it does not manifest in the profile, *i.e.,* if the actions in both tables that cause the dependency are not in any set of non-exclusive actions. To keep changes to the program tractable for the programmer, P^{2}GO removes only a single dependency, even when multiple candidates exist. The programmer can re-run P^{2}GO to successively remove further dependencies.

4.2.2 Removing dependencies

P^{2}GO automatically removes the unseen dependencies by modifying the program to indicate to the compiler that two tables are mutually exclusive. Concretely, it adds a conditional statement such that one of the dependent tables is only applied if the other misses.

4.2.3 Preserving the program's behavior

While the optimization preserves the program behavior for the input trace, there may exist packets in practice for which the dependency manifests, yet no such packet is part of the trace. To avoid this, P^{2}GO returns the pair of tables whose dependency is removed and the observation that no packet can match both. Thus, the programmer can decide if such a packet exists, even though no such packet is contained in the trace.

An alternative approach to deal with inaccurate observations without involving the programmer would be to detect them at runtime. If the first table *hits*, we could apply a new table that matches on the same fields as the second table and triggers a notification to the controller. This approach only detects the problem; we leave mitigation for future work.

4.3 Reducing memory

The goal of this phase is to reduce inefficient resource allocations due to a lack of knowledge of specificities of the underlying architecture, *e.g.,* the available memory per stage. Often, independent tables cannot fit in the same stage as the cumulative memory they require exceeds the available stage memory. In this case, an additional stage will be allocated, which might be barely used, especially if the tables narrowly exceed the available memory. To seize this opportunity, P^{2}GO first probes whether a large memory reduction shortens the pipeline. Next, it finds the minimum required reduction to save a stage. Finally, it verifies that the reduction does not affect the program's behavior.

4.3.1 Identifying and optimizing savings

For each table, P^{2}GO initially halves the allocated memory and compiles the resulting program to compare the number of required stages. If the new program requires at least one stage less, the table is kept as a candidate. Among all candidates, P^{2}GO selects the one with the lowest hit rate, to minimize the risk of impacting the program's behavior. Next, P^{2}GO uses binary search to find the minimum memory reduction that shortens the pipeline.

4.3.2 Preserving the program's behavior

Intuitively, after this modification, new rules added by the operator might no longer fit into the resized table, or there might not be enough register memory to store measurements. To avoid this, P^{2}GO: *(i)* verifies that the memory reduction does not change the behavior of the program on the input trace and *(ii)* returns the change to the programmer. For the former, P^{2}GO verifies that the memory reduction does not change the program profile. For the latter, P^{2}GO returns the resized table or register and its new size. Thus, the programmer can determine whether the change endangers the correctness of the program.

4.4 Offloading code to controller

The goal of this phase is to offload processing to the controller (or other devices), freeing up stages for more useful operations. This opportunity arises when a certain type of traffic is rare or when a feature is barely used. To that end, P^{2}GO first identifies all possible combinations of tables (code segments) that could be offloaded to the controller. Next, it selects the candidate that minimizes the load on the controller.

4.4.1 Identifying offloadable segments

Not all parts of a P4 program can be offloaded. In particular, the offloaded code segment should be self-contained such that packets forwarded to the controller: *(i)* need no additional state (*e.g.,* metadata fields); and *(ii)* need not return to the data plane to complete processing after having been processed in the control plane. Essentially, P^{2}GO considers a code segment as candidate for migration if it does not have any dependency with the rest of the program that will remain in the data plane.

Concretely, a given set of tables and registers S is called valid if and only if all of the following are true:

(i) S is a set of closed directed acyclic subgraphs of the control graph

(ii) There does not exist an element e in the complement of S such that S depends on e or e depends on S. An element e depends on S if and only if e depends on at least one element in S. Similarly, a set S depends on an element e if and only if at least one element in S depends on e. An element e_2 depends on an element e_1 if and only if e_2 utilizes state/data modified by e_1.

(i) is important to avoid having to send packets from the control plane back to the data plane. Instead, the control plane is responsible for forwarding incoming packets correctly. *(ii)* is crucial to avoid split-brain scenarios between data and control planes.

4.4.2 Optimal code segment

Among all candidates, P^{2}GO selects the segment that causes the least traffic to be redirected to the controller while saving at least one stage. P^{2}GO finds this segment across all candidates using dynamic programming. To compute the stage savings and the portion of packets redirected to the controller P^{2}GO compiles and profiles a modified program for each candidate. P^{2}GO automatically generates such a program for each candidate by replacing the corresponding segment with a table that redirects traffic to the controller and by adding match-action rules equivalent to the superset of match-action rules of the candidate segment. [2]

4.4.3 Generating the controller code

Currently, P^{2}GO informs the programmer of the removed tables that need to be implemented elsewhere. Using a recently added compiler backend [6] compiling P4 to uBPFs [5], one could automatically offload data plane sections to, for instance, Open vSwitch [7].

4.4.4 Preserving the program's behavior

If more packets hit the removed segment in practice than during profiling, the load on the controller increases. Thus, more packets will suffer increased delay. This might be especially undesired if the selected code segment is triggered upon a detected anomaly. Indeed, if the trace does not contain abnormal traffic, the corresponding code segment would seem unused. To avoid removing code segments that might be extremely useful in very critical times for a network, P^{2}GO returns to the programmer the code segment that could be offloaded (in table names). The programmer can then decide whether she needs the particular code segment in the data plane.

[2] P^{2}GO's PoC implements a simplified version: the superset of rules is not generated, instead the table's default action is to redirect packets to the controller. While this could forward possibly more packets than intended, it is sufficient to illustrate the concept. All examples in 5 are *not* impacted by this problem.

Chapter 5

Preliminary Evaluation

Another important chapter is the evaluation which should clearly describe the experiments you performed (setup, number of measurements, ...), show the results you achieved in figures and/or tables and discuss and compare the results.

In this section, we use three examples to show that our PoC implementation of P^{2}GO is capable of producing optimized P4 code when presented with traffic traces that allow it. Below, we briefly explain each example and how P^{2}GO optimizes it. P^{2}GO can reduce the number of stages for each example by applying one of its optimizations (Table 5.1). All example programs are written in $P4_{14}$ and run on a Tofino switch [2]. We generated traffic for each example using a traffic crafting library [10]. Excluding compilation time, P^{2}GO's runtime is in the order of tens of seconds. Reducing memory requires re-compilations for the binary search (§4.3), resulting in a total runtime of a few minutes.

5.1 NAT & GRE

We use the features NAT and GRE (tunneling) from switch.p4. These features are dependent, as tunneled packets might need IP address translation after reaching their destination. However, the example traffic trace does not contain any packets making use of both features simultaneously.

By profiling, P^{2}GO observes that the tables of both features are independent and removes the dependency between NAT and GRE. This allows the compiler to place both features in the same stage and effectively saves one stage (see Table 5.1).

5.2 Sourceguard

We implement the Sourceguard feature from switch.p4.[1] Sourceguard ensures that clients only use IPs that were assigned statically or by a DHCP server. Sourceguard checks if the 'source address' of a packet is contained in a DHCP snooping database. We have implemented this Database as a Bloom Filter (BF) with two hash functions.[2]

P^{2}GO observes that if we decrease the amount of memory assigned to the BF, we can gain a stage. P^{2}GO resizes a single register array by merely -8.4%, maximizing memory utilization while saving a stage (see Table 5.1).

[1] We adapted the implementation to be standalone and to use stateful memory for the DHCP database.

[2] We cannot report the exact amount of memory used due to NDA.

Example	Relevant Optimization	Stages	
		Before	After
NAT & GRE	Removing Dependencies	4	3
Sourceguard	Reducing Memory	5	4
Failure Detection	Offloading Code	4	2

Table 5.1: P^{2}GO reduces the pipeline of each example by at least one stage.

5.3 Failure Detection

This example aims at detecting link failures, inspired by Blink [1]. It notifies the controller if prefixes experience more retransmissions than a predefined threshold. Our example includes a BF to detect retransmitted packets, a CMS to count the retransmission per prefix, and a table `FailureAlarm` to push notifications to the controller.

P^{2}GO's profiling shows that only a few packets use the CMS, and even fewer are matched by the table notifying the controller. Indeed, only retransmitted packets use the CMS and `FailureAlarm` matches as often as there are remote failures. P^{2}GO offloads the CMS to the controller and hereby frees two stages (see Table 5.1).

Chapter 6

Outlook

P^{2}GO demonstrates that profiling can reveal ways to *statically* shorten the *physical pipeline* of a P4 program at the *device-level* without changing its behavior. This is only the tip of the iceberg of potential optimization opportunities enabled by profiling techniques. Next, we discuss the possibilities of *dynamic*, *multi-dimensional*, and *network-wide* optimizations.

6.1 Dynamic compilation

P^{2}GO's offline optimizations of a P4 program are beneficial for as long as the computed profile remains representative. Yet, network traffic and device configurations can change over time, leaving an initial profile outdated. A promising research direction to tackle this is *online profiling* in which we would instrument the program with monitoring instructions that update the profile at run time similarly to [20, 25]. Online profiling enables real-time adaptation of the running code by optimizing, compiling, and loading programs at runtime. However, real-time monitoring is computationally expensive, creating a trade-off between profiling accuracy and overhead. Additionally, frequently compiling and reloading a new program to the device might lead to downtimes.

Research question: Where is the sweet spot for maximizing the benefits of online profiling, given the cost of monitoring, compiling, and loading new programs?

6.2 Multi-dimensional optimizations

Programmable data planes tend to be limited across many resources (*e.g.,* stages, buses, ALUs, PHVs), all of which can cause the compilation of a program to fail. In this paper, we show that profiling can be used to optimize the number of required stages. Yet, this approach can be extended to all hardware resources, effectively broadening the optimization space. Navigating this multi-dimensional optimization space is complex, as there might be many different combinations of optimizations whose impact on the program is hard to statically predict.

Research question: How can profiling help the compiler find the modifications that optimize the program while having minimum impact on its behavior?

6.3 Network-wide compilation

While our work shows the potential of profiling for a *single* network device, it also opens the door for profile-guided optimizations in a network-wide context. Oftentimes, a network contains multiple

devices managed by a centralized controller that dynamically installs match-action rules. Optimizing the code in each switch in isolation is *(i)* suboptimal as it prevents us from splitting functionality across devices; and *(ii)* often infeasible as devices might be interdependent. With an appropriate abstraction similar to the "one big switch" abstraction in software-defined networking [18], we believe profiling could also guide network-level optimizations.

Research question: How can we design an abstraction that encapsulates all computational resources and architectural restrictions of (a set of) RMT devices [15] to allow for network-wide profile-guided optimizations?

Chapter 7

Summary

Our work P^{2}GO showcases how static traffic profiling enables various optimization techniques used for minimizing resource usage on programmable switches. We built a self-contained proof-of-concept focusing on three optimization approaches tackling inefficiencies at different levels (architecture, hardware, network) and carried out a preliminary evaluation in a controlled environment. While we focused on statically reducing pipeline stages of one device, we acknowledge and propose promising research directions facilitated through online traffic profiling and how it can fundamentally change the workflow of network programmers.

Bibliography

[1] Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association.

[2] Barefoot tofino. `https://barefootnetworks.com/products/brief-tofino/`, accessed Jun 2020.

[3] Behavioral model version 2. `https://github.com/p4lang/behavioral-model`, accessed Jun 2020.

[4] Clang Compiler User's Manual. `https://clang.llvm.org/docs/UsersManual.html`, accessed Jun 2020.

[5] Linux kernel documentation on berkeley packet filters. `https://www.kernel.org/doc/Documentation/networking/filter.txt`, accessed Jun 2020.

[6] p4c-ubpf: a new back-end for the p4 compiler. `https://p4.org/p4/p4c-ubpf.html`, accessed Jun 2020.

[7] P4rt-ovs: Programming protocol-independent, runtime extensions for open vswitch using p4. `https://github.com/Orange-OpenSource/p4rt-ovs`, accessed Jun 2020.

[8] PROPELLER: Profile Guided Optimizing Large Scale LLVM-based Relinker. `https://github.com/google/llvm-propeller/blob/plo-dev/Propeller_RFC.pdf`, accessed Jun 2020.

[9] Protecting against rogue dhcp server attacks. `https://www.juniper.net/documentation/en_US/junos/topics/topic-map/example-configuring-port-limiting.html`, accessed Jun 2020.

[10] Scapy, packet crafting library. `https://scapy.net`, accessed Jun 2020.

[11] Abhashkumar, A., Lee, J., Tourrilhes, J., Banerjee, S., Wu, W., Kang, J.-M., and Akella, A. P5: Policy-driven optimization of p4 pipeline. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2017), SOSR '17, Association for Computing Machinery, p. 136–142.

[12] Arashloo, M., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. pp. 29–43.

[13] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4:

Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[14] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[15] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, Association for Computing Machinery, p. 99–110.

[16] Cormode, G., and Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. In *LATIN 2004: Theoretical Informatics*, M. Farach-Colton, Ed., Lecture Notes in Computer Science, Springer, pp. 29–38.

[17] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 103–115.

[18] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, Association for Computing Machinery, p. 13–24.

[19] Kanizo, Y., Hay, D., and Keslassy, I. Palette: Distributing tables in software-defined networks. In *2013 Proceedings IEEE INFOCOM* (2013), pp. 545–549.

[20] Kodeswaran, S., Arashloo, M. T., Tammana, P., and Rexford, J. Tracking p4 program execution in the data plane. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2020), SOSR '20, Association for Computing Machinery, p. 117–122.

[21] Moshref, M., Yu, M., Sharma, A., and Govindan, R. Vcrib: Virtualized rule management in the cloud.

[22] Panchenko, M., Auler, R., Nell, B., and Ottoni, G. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019), IEEE, pp. 2–14.

[23] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (2017), pp. 164–176.

[24] Yu, M., Rexford, J., Freedman, M. J., and Wang, J. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, Association for Computing Machinery, p. 351–362.

[25] Zhang, C., Bi, J., Zhou, Y., Wu, J., Liu, B., Li, Z., Dogar, A. B., and Wang, Y. P4db: On-the-fly debugging of the programmable data plane. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (2017), pp. 1–10.