P^{2}GO: P4 Profile-Guided Optimizations

Patrick Wintermeyer
ETH Zürich
patricwi@ethz.ch

Maria Apostolaki
ETH Zürich
apmaria@ethz.ch

Alexander Dietmüller
ETH Zürich
adietmue@ethz.ch

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

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.

ACM Reference Format:
Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P^{2}GO: P4 Profile-Guided Optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425941

1 INTRODUCTION

Thanks to programmable data planes, network programmers can now define the forwarding behavior of their switches using programming languages such as P4 [14]. 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 [24] is a challenging problem that existing P4 compilers approach with various optimizations techniques [18].

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 [3], BOLT [23], Propeller [7]).

Profile-guided optimization × 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 fewer 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 fewer 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 find opportunities for memory optimizations while verifying 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 significant resources. Such segments are good candidates to be offloaded to software.

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425941

Example 1 P4 program. The percentages show the *hit rate* for each table determined by profiling (see §2.2).

```
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
```

Preserving semantics P^{2}GO preserves the semantics of the original program on the provided traffic trace. Yet, if the traffic trace is not representative, the behavior of the optimized program might diverge from the original. P^{2}GO circumvents this problem by directly involving the programmer in the optimization process. Specifically, P^{2}GO reports the adaptations it made to the original program together with the profile-based observations that guided each individual change. The programmer can then choose to selectively accept or reject them based on her knowledge of the general traffic. Exposing those profile-based observations that guided the performed optimization reduces the programmer's burden: she only needs to verify a few particular observations, instead of providing a perfectly representative trace or considering *all* potentially useful properties of the program and traffic.

Novelty Existing P4 compilers assume that the P4 program is static and needs to be mapped to hardware resources without any modifications. P5 [10] 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 address even *implementation-level* inefficiencies.

2 OVERVIEW

In this section, we illustrate with an example how P^{2}GO leverages profiling to optimize P4 programs to use fewer resources. We first describe the example and how the compiler maps it to hardware (§2.1), before describing how P^{2}GO optimizes the process (§2.2).

2.1 Example program

Consider a network programmer working for an enterprise network who gets asked to turn her P4-based IP router into a stateful firewall. Specifically, she needs to make the switch drop: *(i)* UDP packets targeted to specific ports; *(ii)* packets coming from untrusted DHCP servers [8]; and *(iii)* DNS packets once a given amount of queries is reached (128 per source/destination IP, in this example). Coding

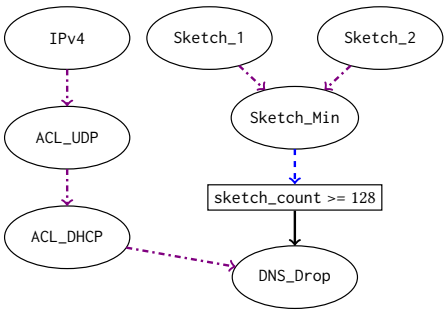

Figure 1: Dependency graph for Ex. 1. Tables are dependent if their actions modify the same fields (dash-dotted violet arrows). A table/control statement depends on another table if it reads a field the latter modifies (dashed blue arrow). A table can also depend on a control statement (black arrow).

this in P4 requires to program the parser to extract the relevant fields from the packet headers, before specifying the control flow of the packets. This control flow defines the processing logic through a sequence of match-action tables. Each table maps (parsed) header fields to a given set of actions. Ex. 1 shows the control flow of our programmer's code, which is composed of 7 tables. Besides the original table used for IP forwarding (`IPv4`), she adds a table `ACL_UDP` to drop UDP packets for specific UDP ports, and a table `ACL_DHCP` to drop DHCP packets based on their ingress port. Third, she counts the number of DNS queries per IP using a Count-Min Sketch (CMS) [16] with two register arrays and two hash functions. The program accesses the arrays with the tables `Sketch_1` and `Sketch_2`. The query count (`sketch_count`) is the minimum of the counts stored in both arrays (table `Sketch_Min`). If the query count exceeds the threshold, packets are dropped (table `DNS_Drop`).

Compilation and dependencies Next, the programmer passes the program to a target-specific P4 compiler. The compiler maps the program to the hardware, namely an ingress and egress pipeline, each composed of several fixed-order stages. The compiler tries to minimize the required stages to make the pipeline as efficient as possible and to allow the program to fit into the hardware. Two main factors prevent the compiler from allocating different tables in the same stage: *(i)* dependencies; and *(ii)* limited stage memory. First, two tables are dependent if their actions either modify the same header or metadata fields, or one action reads a field the other modifies. Ex. 1 contains several dependencies (Fig. 1). For example, `IPv4` and `ACL_UDP` are dependent, as their respective drop actions must set the egress port to a special 'drop' value. Second, each stage has limited memory for tables and register arrays.[1] In our example, both `Sketch_1` and `Sketch_2` require memory for their register arrays and their cumulative size exceeds the memory of a single stage. Thus, the tables need to be placed in different stages.

Compiler output Besides the binary to run in the target, the compiler (typically) provides: *(i)* the actual mapping of the program to the physical stages (Table 2); *(ii)* the dependency graph (Fig. 1); and *(iii)* the control graph, containing all possible execution paths packets may take through the program (not shown).

[1] This is highly simplified due to NDA.

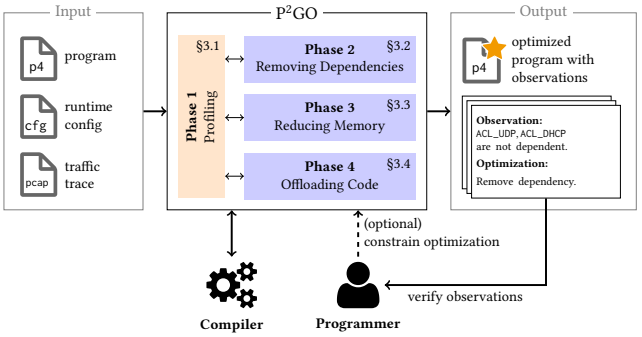

Figure 2: P²GO works alongside the compiler, performs a series of profile-guided optimizations to produce an optimized program which requires fewer resources. P²GO returns the profile-based observations which guided the optimizations and which the operator needs to verify.

2.2 Profile-guided optimization with P²GO

P²GO aims to optimize P4 programs by leveraging profiling techniques (Fig. 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 recompiles 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 a segment of code is seldom used, together with a version of the data-plane code which forwards the corresponding packets to the controller. If the programmer believes these observations are general, she can accept the optimized program and let P²GO optimize the program further. Otherwise, if she suspects that these observations are specific to the input traffic trace and do not generalize to all traffic, the programmer can re-run P²GO with one or more optimizations disabled or with a new traffic trace that is tailored to the particular observation.

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.

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, 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 1 give an example profile for our program.

Phase 2: Removing dependencies In this phase, P²GO compares the dependency graph with the profile to detect dependencies that do not manifest in practice. P²GO observes that while `ACL_DHCP` depends on `ACL_UDP` (see Fig. 1), there is no set of non-exclusive actions that contains the dependent actions of both tables (see Table 1). In other words, these actions are never applied to the same packet. P²GO 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 2). Note that P5 [10] would not be able to remove such a dependency as an operator might need both ACLs.

[2]In practice, some match-action rules may be added by a central controller managing multiple devices, which requires network-wide 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 1: During profiling, P²GO 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.

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 2: With each optimization phase, P²GO reduces the stages required by the example program (Ex. 1). 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.

Phase 3: Reducing memory In this phase, P²GO searches for opportunities to shorten the pipeline by slightly reducing the allocated memory to particular tables. In our example, P²GO first finds that reducing either the memory allocation for table `Sketch_1` or for table `IPv4` also reduces the number of required stages (see §3.3). Starting with `Sketch_1`, as it has a lower hit rate than `IPv4`, P²GO 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. Since the input trace and thus the query count is the same, P²GO concludes that the change is caused by the optimization. Indeed, a CMS is a probabilistic data structure and a reduction of its memory can result in additional hash collisions and thus in over-counting. P²GO discards the optimization of `Sketch_1` as it changed the program's behavior. Following, P²GO considers `IPv4`, finds that reducing its memory does not change the program behavior and applies this modification. The optimized program occupies one stage less (Table 2).

Phase 4: Offloading code to the 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. As a result, 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 a result, the pipeline requires three stages less (Table 2).[3] P^{2}GO reserves code offloading as the last phase to allow optimizing the data plane first. For example, 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 [10] would not remove this segment as it is used.

What if the program does not fit? P^{2}GO can reduce the number of required stages even if the program initially does not fit in the hardware. Concretely, P^{2}GO could compile and profile the program in simulation, independently of the required resources. Furthermore, the compiler generates the dependency graph *before* it tries to find a mapping of the program to the hardware resources. Consequently, all optimization phases which require only profiling and the dependency graph (*i.e.,* 2 & 4) can run without further modifications. In effect, P^{2}GO has the potential to produce an optimized program that fits the hardware.

3 P^{2}GO DETAILED DESIGN

Using the Tofino simulator and compiler (SDE 8.2) provided by Barefoot, P^{2}GO optimizes programs written in $P4_{14}$.[4] We have implemented P^{2}GO in Python using ~300 LoC the for profiling phase and ~1200 LoC for the optimization phases. In the following, we elaborate on each phase.

3.1 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 sets of actions/tables were applied (at least once) to the same packet. In the following, we elaborate on how P^{2}GO instruments the program and creates the profile.

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. Moreover, as each header field is modified in a distinct action, the instrumentation does not introduce new dependencies and cannot increase the programs's required stages.

[3]Observe that removing any single one of the `Sketch*` and `DNS_Drop` tables alone would not decrease the traffic moved to the controller.

[4]We refrained from using bmv2 [2] as our evaluation requires realistic resource allocation on phase 3.

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 on the same packet(s) (non-exclusive actions).

3.2 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. In such tables exist, P^{2}GO modifies the program to explicitly express that they can fit in the same stage, effectively removing the dependency. 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, the programmer would need to consider all possible table pairs, which can be overwhelming and without any guaranteed benefit.

Identifying unseen, yet restrictive dependencies P^{2}GO considers as candidates for removal only dependencies (§2) that are in the longest path of the dependency graph, as only those have the potential to shorten the pipeline. 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. Even if multiple such dependencies exist, P^{2}GO removes only a single dependency at a time to keep changes to the program tractable for the programmer. If needed, the programmer can re-run P^{2}GO to remove further dependencies.

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

Preserving the program's behavior While the optimization preserves the program behavior for the input trace, there may exist packets for which the dependency manifests (even though no such packet is part of the trace). To avoid this, P^{2}GO outputs 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 , reporting the dependency. Still, this approach only detects the problem; we leave mitigation for future work.

3.3 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. If this is the case, it finds the *minimum* required reduction that still saves a stage. Finally, it verifies that the memory reduction does not affect the program's behavior on the provided trace.

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. Binary search allows P^{2}GO to find the minimum reduction without a concrete description of the hardware, *i.e.,* how much (and what kind of) memory is available per stage.

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 facilitate the program's logic. 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 outputs the resized table or register and its new size. The programmer can determine whether the change endangers the correctness of the program.

3.4 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, *i.e.,* the segments that are used by the least traffic.

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 do not need: *(i)* additional state (*e.g.,* metadata) to be processed; and *(ii)* additional processing in the data plane before they are forwarded.

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: *(i)* replacing the corresponding segment with a table that redirects traffic to the controller; and *(ii)* adding match-action rules equivalent to the superset of match-action rules of the segment.

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 [5] compiling P4 to uBPFs [4], one could automatically offload data plane sections to, for instance, Open vSwitch [6].

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 situations, P^{2}GO outputs the code segment that could be offloaded (in table names). The programmer can then decide whether she needs to keep it in the data plane.

4 PRELIMINARY EVALUATION

In this section, we use three examples to show that our proof-of-concept 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 3). All example programs are written in $P4_{14}$ and run on a Tofino Wedge 100BF-32X [1]. We generated traffic for each example using a traffic crafting library [9]. While, P^{2}GO's runtime for profiling and analysis (*i.e.,* excluding compilation time) is in the order of tens of seconds, P^{2}GO might require the program to be (re-)compiled multiple times, which will increase its runtime. In any case, though, this preprocessing is only done once and offline.

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, effectively saving one stage (see Table 3).

Sourceguard We use the Sourceguard feature from switch.p4.[5] 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 using register arrays in the data plane.[6]

P^{2}GO observes that if we slightly decrease the amount of memory assigned to the BF, we can save one stage. P^{2}GO resizes a single register array and reduces its size by merely 8.4%, maximizing memory utilization while saving a stage (see Table 3).

[5] We adapted the implementation to be standalone and to use stateful memory for the DHCP database.
[6] 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 3: By leveraging profiling, P^{2}GO shortens the pipeline of each example by at least one stage.

Failure Detection This example aims at detecting link failures in the data plane, and is inspired by Blink [17]. The switch notifies the controller if some 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 3).

5 RELATED WORK

Profiling execution paths Previous work has explored tracking packets in the data plane's control logic to provide visibility to the programmer. P4DB [26] uses match-action tables and packet digests to report a packet's path to a debugging platform. Another approach [21] allows online tracking of packets in the data plane through a variation of the Ball-Larus encoding. While simpler, P^{2}GO's offline profiling approach is adequate to guide our optimizations. Future work could pair such online profiling with P^{2}GO's optimization phases to allow real-time optimizations (see §6).

Optimizing code for programmable switches There exist numerous approaches to optimizing the placement of SDN policies through virtualization and network abstraction [19, 20, 25]. These works optimize the number and placement of rules needed to enforce policies, relying on a global view of the network. Similarly, [22] focuses on rules, applied to both hypervisors and switches. SNAP [13] implements a new high-level programming language to efficiently distribute state, processing, and routing amongst devices of a network. Unlike previous works, P^{2}GO operates on a per-device level and on the RMT architecture [15]. Particularly, P^{2}GO directly optimizes P4 code, which is more expressive *i.e.,* supports stateful memory, metadata, programmable parsing.

P5 [10] also optimizes P4 code by removing dependencies between high-level features if they do not manifest in the high-level policies provided by the programmer. On the contrary, P^{2}GO relies on information discovered during profiling and removes dependencies on a finer-grained manner *i.e.,* individual actions and tables. Finally, unlike P5, P^{2}GO also considers target-specific memory optimizations and offloading code segments.

Profile-guide optimizations for general-purpose programming languages Industry-grade compilers and tools [3, 7, 23] use profile information mostly to optimize code layout, inlining, and register allocation, while some research compiler frameworks employ profile information to guide speculative optimizations [11, 12].

6 FUTURE RESEARCH AGENDA

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 observed behavior. This is only the tip of the iceberg of potential optimization opportunities enabled by profiling techniques. In the following, we discuss the possibilities of *dynamic*, *multi-dimensional*, and *network-wide* optimizations.

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 runtime, similar to [21, 26]. Online profiling enables real-time adaptation of programs by optimizing, compiling, and loading them 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 and dynamic compilation, given the cost of monitoring, compiling, and loading new programs?

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 across multiple dimensions while having minimum impact on its behavior?

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 [19], we believe profiling could also guide network-level optimizations.

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

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their helpful feedback. This work was partially supported by ETH Research Grant ETH-03 19-2.

REFERENCES

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

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

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

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

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

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

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

[8] 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.

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

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

[11] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 351–367, New York, NY, USA, 2020. Association for Computing Machinery.

[12] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August. Scaf: A speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 638–654, New York, NY, USA, 2020. Association for Computing Machinery.

[13] M. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. pages 29–43, 08 2016.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

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

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

[17] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. 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.

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

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

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

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

[22] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Vcrib: Virtualized rule management in the cloud. 01 2013.

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

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

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

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