Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Gauging Risk in Resource Optimizations on Stateful Packet-Processing Devices

Master Thesis

Author: Patrick Wintermeyer

Tutors: Maria Apostolaki, Alexander Dietmüller, Edgar Costa Molero

Supervisor: Prof. Dr. Laurent Vanbever

April 2021 to September 2021

Abstract

Resource constraints on programmable switches are a serious challenge when developing networking applications. Recent research has explored different avenues to tackle this problem on stateful devices using traffic insights and semantically non-preserving optimizations. However, these methods either involve the programmer to supervise and approve (speculative) optimizations or are very conservative in their optimization approaches.

In this work, we attempt to bridge the aforementioned gap by systematically assessing the impact of resource optimizations on the semantic equivalence of stateful programs. We present a novel framework to classify optimizations with respect to their packet correctness and state consistency, apply our method to two state-of-the-art optimizations, and build a prototype classifier using symbolic execution.

Contents

Chapter 1

Introduction

Hardware resource budgets on networking switches are notoriously tight. Resources such as memory (SRAM, TCAM), pipeline stages, and arithmetic logic units (ALU) are common limiting factors when deploying networking applications. However, the emergence of software-defined networking (SDN) and OpenFlow [37] has enabled new resource optimizations. To alleviate memory allocation bottlenecks, commonly faced for large forwarding tables, abstraction and offloading mechanisms have been proposed in the past [27, 51, 30].

Fast forward to the day and age of fully programmable switches: while the programmer's flexibility has massively increased, the underlying resource scarcity still persists. Thus, more recent work on P4 switches [10] has proposed leveraging policy [1] or traffic insights (P2Go [48]) to perform more aggressive optimizations. Utilizing traffic profiles, P2Go resizes underutilized memory allocations or selectively removes unseen dependencies in a P4 program by changing the control flow. Memory allocations can equally well be used for table entries or stateful data structures. Dependencies are removed by marking code sections as mutually exclusive, which increases parallelism on switches. Since these optimizations are not semantically preserving, P2Go ultimately relies on the programmer to approve all changes.

However, this puts a lot of responsibility on the programmer. Semantically non-preserving changes can not only lead to wrong forwarding behavior of selective traffic, if a rule in a resized forwarding table is missing because of a prior optimization. It can also result in the permanent corruption of stateful data structures. This corruption can then propagate internally and lead to misbehavior of the overall switch, effectively breaking correct packet processing of all traffic.

In this work, we explore the space of resource optimizations in the context of semantic equivalence. By refining the notion of semantically preserving optimizations and introducing a separation between packet processing operations and state modifications, we attempt to better understand the effects of aggressive resource optimizations on P4 switches.

Our main contribution is a framework to classify optimizations in terms of their impact on program correctness (Section 2). Furthermore, we apply this model to two types of state-of-the-art optimizations and propose appropriate mitigations for misspeculations (Chapter 3). In Chapter 4, we implement a prototype classifier and show the practicality of our approach by evaluating the classifier on real-world examples (Chapter 5). Finally, we discuss future research avenues to expand our model to stateful, dynamic, heterogeneous networks (Chapter 6).

Chapter 2

Framework for Correctness Guarantees

In the following sections, we present a framework to classify a given optimization on a P4 program with respect to semantic correctness. To do so, we refine the notion of correctness and divide it further into *packet correctness* and *state consistency.* We give an intuition of the core insights of our framework before applying its concepts to two state-of-the-art optimizations (see Chapter 3).

2.1 Architecture

SDN packet processing was revolutionized by the RMT architecture [11] and its flexibility with respect to match-action pipelines and header extractions. Its *stateful* successor architecture model is the Portable Switch Architecture (PSA) [23]. Specifically, it extends RMT by adding support for stateful data structures, such as registers, to allow access to persistent memory across packets. Our model targets PSA and optimizations for PSA, which also includes optimizations for both software simulators(*e.g.,* bmv2 [40]) and hardware devices (*e.g.,* Tofino [6]).

State. We refer to stateful data structures such as registers, meters, and counters as *state.* Registers are the most versatile data structure of all stateful data structures available in P4, as there is practically no restriction on the data stored in them. Indeed, one could implement a form of meters and counters purely with registers (however, the interface access control will be more permissive than with meters and counters). Therefore, without loss of generality, we use state and registers interchangeably.

Note, that we do not consider tables when we refer to state. While this might seem counter-intuitive at first, tables cannot be updated through the data plane. The data plane can merely send a packet to the controller which will then take appropriate action. Furthermore, tables are generally infrequently updated compared to registers and can be synchronized across devices more easily.

State Modifier. A *state modifier* M is a function, that transforms an initial state state S_i to a final state S_f. The transformation is triggered by packet p and given table contents (action arguments) A:

$$S_{final} = M(S_i, p, A) \tag{2.1}$$

Since the data plane cannot modify table entries and arguments, they can be seen as static or constant, and we will use the shorter notation $S_{final} = M(S_i, p)$.

In contrast to state modifiers, state can also be *read-only*, in which case state is only accessed to retrieve values, but not to write values to state. We refer to state S as being read-only if there does not exist a state modifier for S.

2.2 Packet Properties

In this section, we define packet properties and behavior that can arise in the context of program optimizations.

Packet Reordering. Given two packets p_1 and p_2 reaching the switch at times $t_{i,1}$ and $t_{i,2} = t_{f,1} + \tau$ respectively, where $\tau > 0$, *post-processing reordering* corresponds to a reordering of p_1 and p_2, after they have been processed correctly in order of arrival. In other words, it is as if p_1 had suffered a delay $\delta > \tau$ in the final egress buffer, such that

$$\begin{aligned} t_{f,1} &= t_{i,1} + t_{processing} + \delta \\ t_{f,2} &= t_{i,2} + t_{processing} \quad < \quad t_{f,1} \end{aligned} \tag{2.2}$$

We define *pre-processing* analogously. Specifically, once a packet has entered the processing pipeline, no further reordering inside the switch is allowed.

Misspeculated Packet. A misspeculated packet p_x traversing the control flow of an optimized program H' is a packet that does not adhere to the assumptions of the optimization O. In other words, it is transformed incorrectly with respect to the original program H, because the optimization's requirements do not hold for p_x, such that:

$$H(p_x) \neq H'(p_x) \tag{2.3}$$

For example, if an optimization removes a dependency between two tables, then a packet p_x that would justify the dependency between both tables is a misspeculated packet.

2.3 Framework Design

We decompose an optimization into its effects on stateful and non-stateful components of a P4 program. Specifically, we focus on the consequences of a misspeculated packet p_x both on its own correctness and on the correctness of state. Given an optimization O and p_x, one can roughly divide this space into four scenarios:

1. Correct packet processing, correct state modification
2. Wrongful packet processing, correct state modification
3. Correct packet processing, wrongful state modification
4. Wrongful packet processing, wrongful state modification

In order to perform this classification, we need to consider all possible structures of P4 programs — a very large space to explore. Thus, for a given optimization, we focus on patterns of state usage in P4 programs to classify each pattern with respect to previously mentioned scenarios: where is state used in the program? How does its usage relate to the optimization? What are the worst-case scenarios that could arise from any given (misspeculated) packet?

Because of the complexity of this multi-dimensional problem, the application of this framework to a given optimization is often complicated and non-exhaustive. As with all optimizations, it is a trade-off between generality, applicability, and payoff. However, the core notions introduced here provide a systematic approach to navigate this complex space and give enough flexibility to adapt its concepts to any given optimization.

In the following, we define the terminology of our framework. It builds the foundation for the classification of optimizations in sections 3.1 and 3.2.

State Consistency. As an intuition, state consistency means that all stateful data structures in the switch have either been transformed correctly or not been transformed at all. In other words, at no time will only part of the state be transformed correctly by any packet. Therefore, given a misspeculated packet p_x, subsequent packets $p_{X+1}, p_{X+2}, \ldots$ see the state as if p_x had been reordered pre-processing (state unaffected by p_x) or p_x will seem reordered post-processing (following packets have state from after processing of p_x but are emitted from the switch before p_x).

Given a program H with state S and H's optimized version H' with initial state $S_i = S'_i$, H' is *state-consistent*, if and only if, at all times, modification of S_i and S'_i by any given packet p are identical or S' remains unchanged. If M and M' are state modifiers of H and H' respectively:

$$\forall p, M'(p, S_i) = M(p, S'_i) \vee M'(p, S'_i) = S'_i \tag{2.4}$$

We can then say that optimization O, such that $O(H) = H'$, conserves *state consistency.*

Packet Correctness. Given a program H and its optimized version H', H' is *packet-correct*, if and only if, modification and forwarding of any given packet p by H and H' are identical, given initial states $S_i = S'_i$:

$$\forall p, H(S_i, p) = H'(S'_i, p) \tag{2.5}$$

We can then say that optimization O, given by $O(H) = H'$, conserves *packet correctness.*

Semantic Equivalence. Given a program H and its optimized version H', H' is *semantically equivalent* to H, if and only if, H' conserves both packet correctness and state consistency.

Limitations. We require that the program H does not make any assumptions about the order in which packets reach the switch. This is a reasonable assumption because in general, packet reordering is a common phenomenon and can even happen under normal operation of networking devices [9, 22]. We furthermore clarify that only *pre-processing* or *post-processing reordering* is allowed. Similarly, we assume that the network and applications running on the network support out-of-order processing of packets.

Chapter 3

Application on State-of-the-Art Optimizations

In this chapter, we apply the previously defined framework for correctness guarantees on two state-of-the-art program optimizations: memory reduction and dependency removal. Both optimizations attempt to alleviate common resource bottlenecks by making additional assumptions about the network traffic. Through an in-depth analysis of both optimizations on the effect of misspeculated packets, we present state and packet correctness guarantees.

3.1 Memory Reduction Optimization

Memory on stateful networking switches is a scarce resource. Specifically, ternary content-addressable memory (TCAM) is a common bottleneck in networking applications, because it is oftentimes used for longest-prefix matching (LPM) in forwarding tables. As such, TCAM is a prime candidate for memory optimization techniques. The purpose of memory optimizations such as presented in [48] is to free up unused memory by shrinking table allocations. Should it become necessary to add more rules in an optimized table, a recompilation of the program is needed, or semantic equivalence is at risk.

In this section, we propose a more dynamic approach: instead of simply reducing table allocations, we offload rules to a remote device with caching capabilities. This allows the dynamic placement of rules on either the local switch or on the remote cache. However, not all rules are made equal, because of stateful data structures, that are only available locally. Additionally, TCAM rules are particularly tricky to handle, because of inter-dependencies between different rules — one cannot simply offload a given rule R_x, without considering which other rule would be matched instead if R_x were absent.

Offloadability Problem. Given a TCAM rule R_x and its associated action A_x in a table T, is it possible to offload R_x to a remote processing unit without affecting the correctness of a packet p_x that would match R_x and without affecting the consistency of state S ?

By applying our previously defined framework for stateful data structures and correctness guarantees on the example of memory optimizations, we provide a systematic approach to classify rule offloading with respect to packet correctness and state consistency.

3.1.1 Related Work

Semantically preserving optimizations on static random-access memory (SRAM) and TCAM have been studied extensively in the field of SDN before stateful switches came around. Our framework is orthogonal to these studies and strictly needed for optimizations on stateful switches. For completeness, we mention previous work on network virtualization, single-switch in-place optimizations, and remote cache offloading.

Previous work has proposed approaches to optimize the placement of SDN policies, by relying on a global view of the network. The "One Big Switch" Abstraction [27], Palette [28] and Difane [50] fall into this category of virtualization and network abstraction. However, we focus on a single device in the network.

In Devoflow [15], the authors use a form of "explosion" of TCAM rules to convert them into SRAM rules and thereby lifting the TCAM bottleneck. Indeed, by replacing wildcards in TCAM with several SRAM rules spanning the entire set of possible values (cover set), the semantic equivalence of both tables can be preserved. The inverse operation from SRAM to TCAM is of course also possible, given TCAM availability and that the grouping of SRAM into TCAM rules can be done efficiently. This duality can be leveraged to broaden the applicability space of our framework.

In TCAM Razor [38], the authors use a multitude of minimization approaches to "squash" TCAM rules using decision trees, dynamic programming, and redundancy removal. For example, instead of matching on a set of fields F, they find a cover set F' that uses fewer TCAM rules while being semantically equivalent. This space has been studied extensively by [3, 18, 34, 17, 47] in the setting of stateless programs and can be used *after* the classification by our framework to optimize rules by their offloadability type.

In CacheFlow [30], N. Katta et. al presented an incremental rule-dependency analysis toolkit for TCAM, used to offload TCAM rules to remote caches, based on novel cache replacement strategies. Their key contribution is the splitting of dependency chains efficiently to reduce rule updates and traffic sent to the cache. Flowcache [42] and FreeCache [32] build on top of CacheFlow and explore different metrics to optimize dependencies using cover sets and hash table lookups. However, none of them have studied the impact of rule offloading to remote caches on semantic equivalence, more specifically on state consistency and packet correctness.

3.1.2 Offloadability Classification

We identify five different offloading "types" for table rules, based on various conditions that differ in how exactly packets must be offloaded to ensure packet correctness and state consistency. The existence of state S in the program H and its access type is central to the question of *offloadability*. Each type offers different guarantees with respect to state consistency and packet correctness.

In the following, we first describe the general design of the offloading system, before classifying a given rule R_x in a table T by verifying a set of conditions. For each classification type, we specify guarantees with respect to state consistency and packet correctness and necessary modifications to the program H. Note, that the question of which rule should be offloaded is orthogonal to the classification itself and extremely difficult to answer in the general case (see Section 6.2).

General Design

In the following sections, we will first describe the general setup of a remote cache and its functionality. Then we present the rule reduction on the switch and give a code example that will guide us through each offloadability type.

Remote Processing Unit. To offload rules to a different entity on the network, we assume a software caching system similar to the one described in FlowCache [42] is in place. In essence, the remote cache holds a copy of all tables and their entries in memory. Specifically, the cache holds $T_{complete}$, the full version of optimized table T, with its offloaded entries $\{R_x, R'_x, R''_x, \dots\}$.

Table T

$N°$	`IPsrcAddr`	action argument a	statement performed	offloadability type
1	172.16.0.1	×	`drop_packet = true;`	A
2	192.168.0.1	2	`egress_port = a;`	B
3	10.0.0.1	1	`index = a;`	C
4	10.0.1.1	2	`index = a;`	D
5	10.0.1.2	2	`index = a; TTL=TTL-1;`	D
6	10.1.0.0	0	`index = a;`	E

Table T_{state}

$N°$	`IPsrcAddr`	statement performed
7	192.168.0.1	`IPdstAddr = read(register, index);`
8	10.0.0.1	`IPdstAddr = read(register, index);`
9	10.0.1.1	`write(register, index, IPsrcAddr);`
10	10.0.1.2	`write(register, index, IPsrcAddr);`
11	10.1.0.0	`write(register, index, IPsrcAddr);`

Table 3.1: Example program used to illustrate offloadability types. The offloadability type column is the result of applying the rule classification and not part of the actual program.

Rule updates from the controller are sent to the data plane and the cache, which then update their tables accordingly.

Rule Reduction. Once we identified a set of rules $R = \{R_x, R'_x, R''_x, \dots\}$ of same offloadability type, we introduce a new rule R_m matching exactly like the union of offloaded rules R. The existence of R_m allows us to differentiate between a misspeculation (cache hit, hit on R_m) and an actual miss in T. This is vital because we may have to perform different operations on a cache hit to fulfill certain guarantees. In the general case, more than one rule may be needed to cover R. Without loss of generality, for simplicity purposes, we assume that one rule R_m is sufficient to cover R. As shown by Richard Karp in [29], the calculation of a minimal cover set for R is an NP-hard problem. In the following, we will refrain from investigating efficient algorithms for this purpose and refer to previous work [38]. It would not be necessary to create an exact cover of R; however, we would have to consider R's rule dependencies, similar to [30]. This could greatly reduce offloadability opportunities if any of the dependencies have an offloadability type with higher precedence.

We also introduce a new action A_m, associated with R_m, whose statements depend on R and its offloadability type. A_m is used for adequate packet processing, laid out below for each offloadability type. These modifications require one initial patch and recompilation of the program H (to add action A_m).

Example Program. To illustrate the classification, we will refer to a sample program given by two tables T and T_{state} that are applied in this order. We will classify different rules in T while keeping a close eye on state modifiers in T_{state}. Assume for simplicity purposes, that both tables perform *exact* matching. Table 3.1 shows the entries in both tables and the respective actions performed. Note that for comprehensibility purposes, we make abstraction of actions and directly use statements. Different statements mean that different actions are linked to the respective entries. The first column in each table is not part of the actual program and simply refers to a rule number.

Offloadability Type A

Offloadability type A is characterized by the absence of state accesses after T in the control flow graph.

To fulfill this condition, there are two possibilities: either there is no state access in any control flow path after execution of T, or the state access is conditional (condition or table match) and we can guarantee that for any packet having matched R_x and given rules of all tables, the state access will not be executed. For example, a packet p_x that matches entry 1 will not reach any stateful data structure, because it will be dropped.

Packet processing. When a match on R_m happens, we proceed as follows: set the output port of p_x to the port of the controller and skip further processing in the ingress and egress pipelines (using `skip` and `exit` constructs in P4). Append a digest of all metadata from the data plane to p_x. In the controller, we apply $T_{complete}$ and continue the processing of p_x like in the data plane. Since the controller holds an exact copy of all tables from the data plane and we sent all relevant information to the controller (p_x and metadata), we can guarantee semantically equivalent processing of p_x.

Packet correctness. We can guarantee packet correctness. Packet correctness is not endangered by remote processing, because all remote operations only take as input packet headers or metadata. Both can be transferred to the controller. However, p_x will suffer a significant delay. This corresponds to post-processing reordering.

State Consistency. We can guarantee state consistency. Since state is only used prior to T, it is independent of whether T was applied correctly or not, and thus state will always be consistent.

Offloadability Type B

Offloadability type B is defined by reachable state S_x after T, whose access is independent of the existence of R_x.

Dependency here means that either the state access depends on a condition C_x, which in turn depends on the existence of R_x, or that the state access itself depends on the match on R_x and execution of A_x. Note, that this is different from a static dependency between S_x/C_x and T as reported by the compiler. We are interested in the *effect* of the existence of R_x in T. This requires configuration analysis during rule insertion. Please also note that we define this dependency *transitively*: there can exist a "proxy" variable written to in A_x and read from later during state access.

For example, assume we were to remove entry 2. Then, the egress port of p_x would no longer be set; however, the state access in rule 1 of T_{state} would still be applied correctly. Indeed, rule 2 does not entail a modification of `index`.

An absence of R_x could mean that p_x might match another rule R_y instead (so called rule-shadowing). The associated action A_y could present a dependency with C_x. However, if we offload R_x we add a rule R_m that replaces all R_x with similar properties (see 3.1.2). Therefore, we can exclude the case that a rule R_y gets matched in absence of R_x. Thus, we only need to check if there is a dependency between *the existence of* R_x, specifically A_x, and S_x (or C_x if it exists).

Packet processing. To ensure that all state accesses happen correctly (reads or writes), we send p_x to the controller at the end of the pipeline. When a match on R_m happens, we set the egress port of p_x to the port to the controller. Should any other statement later in the control flow be able to overwrite the egress port, we set a flag in A_m instead and add a simple if-condition on the flag at the very end of the control flow to ensure that the packet is sent to the controller.

Packet correctness. Packet correctness cannot be ensured in the general case. Because of the replacement of the execution of A_x by A_m in which we simply set the egress port, header and metadata modifications of A_x are no longer executed. However, p_x continues its path in the control flow, thereby possibly matching wrongfully on some table T_w and executing (wrongfully) an action A_w. These modifications are not reversible later on, specifically if the mapping between rules and modifications in T_w is not invertible. For a more extensive analysis of the problem and exploration of possible solutions, see Appendix A.1.

State consistency. We can guarantee state consistency since all read and write accesses are independent of R_x. Therefore, by offloading R_x, S_x remains unchanged.

Existence of such opportunities. Can S_x or C_x not depend on R_x and yet be placed after T (aside from hardware constraints)? In other words, can this offloadability type even exist?

In the simple case, assume that there is an action that justifies this dependency, but there is no rule associated to it. Then, the compiler will report this structure as a read-after-write dependency and use two different stages, effectively placing T before S_x. In the more general case, consider a scenario where a rule R_1 is responsible for the dependency with S_x. However, if there is no rule dependency between R_x and R_1, then S_x does not depend on the presence of R_x, despite the existence of a rule that justifies the dependency between T and S_x.

Offloadability Type C

For offloadability type C, we specify that there is read-only state S_x after T, which is dependent on the existence of R_x.

Verification of this condition is fairly simple, as state modifiers (`register_write` in P4) are syntactically easily distinguishable from packet modifiers with state access (`register_read` in P4).

In the example, assume we were to remove the execution of the action linked to rule 3. Then, `index` would not be set to 1 and the subsequent match on rule 8 would effectively set `IPdstAddr` to a wrong value. State is therefore not affected by rule 3, but the correctness of p_x can no longer be guaranteed.

Packet processing. A_m sets the output port of p_x to the port of the controller and skips further processing in the ingress and egress pipelines (using `skip` and `exit` constructs in P4). This allows the controller to keep track of the number of packets affected by this offloadability type's misspeculation and dynamically optimize the number of misspeculations (which rules are offloaded).

Packet correctness. We cannot guarantee packet correctness. Indeed, there exists a read-only state access, dependent on R_x. This means that the read access on state will be incorrect upon misspeculation. The values read from state will eventually incur a modification of p_x with wrong values, either through direct modifiers of the packet or through incorrectly evaluated conditional statements. These modifications cannot be reversed later on, since we do not have access from the remote cache to consistent, timely snapshots of state in the data plane. We cannot correct p_x after the misspeculation.

State consistency. We can guarantee state consistency because S_x is not modified depending on R_x. As such, a misspeculation does not alter state S_x.

Offloadability Type D

For this offloadability type, we leverage the somewhat arbitrary grouping of statements into actions by the programmer. Indeed, splitting up actions into their actual statements (through an intermediate representation) and then regrouping them differently (sometimes even at the cost of duplicating code) is a popular approach used in both traditional compiler optimizations (LLVM [36]) and network abstraction tools such as Lyra [21] and Lucid [44].

Conditions. There is state S_x that is written to dependent on R_x. Additionally, we require, that it is possible to group a set of rules or actions that have the same effect on state, but differ in effect on packet headers, in a new rule-action-pair $R_m \leftrightarrow A_m$.

For example, consider rules 4 and 5. While their actions are different, they set the state relevant variable `index` in the same way and the state access in T_{state} is also equal. Therefore, one could merge both entries. However, this would entail wrong processing of packets matching rule 5, because `TTL` would not be reduced by 1.

Condition verification. To find an appropriate set of actions whose stateful operations can be regrouped in a new action A_m it is not sufficient to simply consider the actions themselves. Instead, we need to navigate a two-dimensional space of action statements and table rules. Indeed, we consider actions and their input arguments (coming from rules) and evaluate their equivalence with respect to state. In other words, regrouped actions only need to be equivalent with respect to state under the set of possible action arguments defined by rules. Indeed, it can even be possible that two rules $\{R_x, R'_x\}$ are linked to the *same* action A_x, but provide different action arguments. However, if the action arguments relevant for stateful write access are equal, we can ensure that the state modification is correct upon misspeculation and therefore merge $\{R_x, R'_x\}$ without even considering other actions in that table or generating a dedicated action A_m — reusing A_x is sufficient.

Packet processing. We statically add the action A_m to contain all necessary stateful operations to ensure the correctness of state. While it is possible to change rules dynamically, actions cannot be changed on the fly and need recompilation. Therefore this type of offloadability is at the intersection of dynamic and static optimizations. We set the output port in a way to send p_x to the controller and let p_x continue on its path in the pipeline. Again, the controller can dynamically regroup rules, although this might require recompilation to change statements in A_m. We could also create actions $\{A_m, A'_m, A''_m, \dots\}$ that contain all combinations of action groupings and then simply add at runtime the correct rule groupings with one of $\{A_m, A'_m, A''_m, \dots\}$.

Packet correctness. Because of how the action A_m and rule R_m are constructed, we neglect packet header and metadata modifications in favor of correct state modification. Therefore, we cannot guarantee packet correctness in the general case.

State consistency. We can guarantee state consistency because through A_m we can ensure that state is modified in the same way as if A_x had been executed. In other words, A_m and A_x are semantically equivalent with respect to state S_x and all regrouped rules R.

Offloadability Type E

Any rules that do not fulfill the conditions of offloadability types A-D, cannot be offloaded without endangering state consistency and potentially impacting large portions of traffic. As such, we recommend keeping rules that do not fulfill any of the conditions above in the data plane.

For example, rule 6 does not have the same effects on the state relevant variable `index` than the other rules. As such, it cannot be merged with other rules, given the current table entries, because we can no longer guarantee state consistency.

Offloadability Precedence

A given rule R_x may be classified as multiple offloadability types simultaneously, because of the existence of multiple stateful data structure accesses. We employ an ordering between offloadability types to set the overall offloadability Ω of R_x using an invertible mapping function $f : \{A, B, C, D, E\} \to \mathbb{N}$, such that:

$$f(A) < f(B) < f(C) < f(D) < f(E) \tag{3.1}$$

Given a program H with n stateful data structure accesses after T and a rule R_x with offloadability types ω_i, $i \in \{1, \ldots, n\}$, R_x's overall offloadability is given by:

$$\Omega = f^{-1}\big(\max_{i \in \{1,\ldots,n\}} f(\omega_i)\big) \tag{3.2}$$

3.2 Dependency Optimization

Data dependencies in pipelined architectures, such as PSA, occur when two tables contain actions that modify or read the same variables; in P4 this corresponds to packet headers and metadata fields, or registers. Hardware constraints limit the number of dependencies in a program: if it shows too many dependencies, the compiler will fail to map it to the switch. As such, state-of-the-art optimizations [48, 1] also attempt to reduce data dependencies. They leverage insights about the network traffic to speculatively remove dependencies that do not manifest.

In this section, we investigate the effects of a *misspeculation* and provide a systematic approach to detect and mitigate misspeculated packets in removed dependencies.

Dependency Optimization Misspeculation

In the context of dependency optimizations, we can refine the definition of a *misspeculated packet* given in 2.2:

Given a program H with a dependency Δ between two tables A_H and B_H, its optimized version H' (with tables $A_{H'}$ and $B_{H'}$ resp.) where the dependency has been removed through optimizations (see 3.2.1), packet p_x scoring hits on both A_H and B_H is a misspeculated packet with respect to H'. Indeed, the optimization was applied because it was assumed, that no packet hits both tables A_H and B_H. The misspeculated packet p_x does not adhere to this assumption. Because of the removal of Δ in H', p_x can now no longer hit both $A_{H'}$ and $B_{H'}$.

Given H', can we put in place a detection hook for p_x to notify the controller of a misspeculation without re-introducing Δ? Can we implement mechanisms to provide packet correctness and state consistency despite the removal of Δ?

Data Dependencies

While there exist different forms of data dependencies, our main focus is on *action dependencies*. Both reverse-read and control flow dependencies do not typically require an additional hardware stage on PSA. Given a program H and two tables A and B applied successively, two actions a_1 and b_1 of A and B (resp.) and a variable v, an action dependency is given by either:

- a statement in a_1 writing to v and a statement in b_1 reading from v (*read-after-write dependency*), or
- a statement in a_1 writing to v and a statement in b_1 writing to v (*write-after-write dependency*)

3.2.1 Related Work

Previous work in P5 [1] and P2Go [48] has shown the potential to reduce dependencies by leveraging high-level policies or traffic traces. However, they rely on the programmer to supervise and approve dependency removal. This puts a lot of responsibility on the programmer: a misspeculation can have dire consequences for both packet correctness and state consistency. Through persistent state over several packets, the whole traffic is at stake. In this section, we provide guidance to the programmer by presenting detection and mitigation mechanisms for misspeculated packets due to dependency optimizations.

3.2.2 Detection of Misspeculated Dependency Removal

The idea to detect a misspeculation was introduced in P2Go. In Program 1 we present the general structure of an optimized program H' with respect to an action dependency Δ previously existing between `table_A` and `table_B`. The `on_miss` statement effectively removes the dependency Δ, while a newly introduced `on_hit` structure serves as detection mechanism through a table `table_fix`. Note, that this part does not introduce new dependencies as long as `table_fix`'s actions do not modify or read the same data structures as table `table_A`.

We construct `table_fix` as follows: `table_fix` matches on the same keys and contains the same entries as `table_B`. `table_fix` has a default action `notify_controller()` that sends the packet to the controller and terminates the control flow execution for p_x to avoid wrongful processing.

Code Snippet 1 Showcasing the control flow of a program with an optimized action dependency between `table_A` and `table_B` in orange, alongside the mitigation hook implemented through `table_fix` in blue.

```
control
    apply table_A :
        on_miss:                              ▷ dependency optimization
            apply table_B
        on_hit:                               ▷ detection & mitigation hook
            apply table_fix
    end apply
...
    apply Y                                   ▷ state modifier
end control
```

3.2.3 Mitigation of Misspeculated Dependency Removal

While detection is the first step towards automatic resolving of misspeculations, it requires external interaction to resolve the problem. We will go one step further and investigate to what extent `table_fix` can be enhanced by additional actions to ensure state consistency.

Identification of State Modifiers

During a misspeculation, it will not be possible to execute `table_B`. Thus, to guarantee state consistency, we need to answer the central question: Does any state modification Y depend on the execution of `table_B`?

Note that we consider dependencies transitively. To answer this question, we distinguish two cases:

- There is **no** state transformation Y such that Y depends on the execution of `table_B`. Then, independently of the execution of `table_B`, state is transformed correctly.
- There **is** a state transformation Y that depends on the execution of `table_B`. Dependencies in general can manifest in the following forms:
  - Control Flow Dependency: There is a semantic dependency between Y and `table_B`, but no data dependency. In other words, this is a constraint on the order in which the packet is applied to tables. Therefore, Y does not depend on the execution of `table_B`.
  - State Reverse Read Dependency: `table_B` reads state that Y modifies. p_x will not be able to read the state, resulting in missing/wrong headers in p_x. However, as by definition `table_B` only *reads* state, the transformation Y does not depend on `table_B`. The local state will remain consistent independent of the execution of `table_B`.

- State Action Dependencies: There exist two subgroups of such dependencies:
  - State Action-Write-After-Write: `table_B` and Y modify the same state. This is impossible because state cannot be accessed/shared across stages.
  - State Action-Read-After-Write: `table_B` reads state and writes to metadata, which is then, in turn, read from and written to state by Y. Another scenario is that `table_B` writes to state and saves the result in metadata, which is then read later on. As such, the important operation here is the writing to state by `table_B` as in the latter scenario there is no modification of state after `table_B`.
- State Match Dependency: `table_B` writes to metadata that is read in C to apply Y. Here, C can be a table, a conditional statement or an independent conditioner (table, condition, switch statement) whose action is Y.

In short, only state match dependencies and state action-read-after-write dependencies are relevant for state consistency. In other words, consistency of state is endangered, if either any value from `table_B` is read to modify state or `table_B` itself modifies state.

Ensuring State Consistency

In order to ensure state consistency despite a misspeculation by p_x, we need to perform the necessary operations to ensure the correct execution of state modifier Y in `table_fix`. Note, that we cannot introduce new dependencies between `table_A` and `table_fix`. This leaves little room for fixes. The question becomes therefore if there is a dependency between state-relevant modifiers (fields necessary for the correct execution of Y) and `table_A`.

If there is no such dependency, we can perform the necessary operations in `table_fix` to ensure correct execution of Y, without introducing a new dependency between `table_A` and `table_fix`. Conceptually, the programmer made a somewhat arbitrary grouping of statements into actions, while we now perform a grouping with respect to state modifiers.

Should such a regrouping not be possible, we would have to revert the already modified state in order to guarantee state consistency. However, such a "rollback" or "reversing of operations" without affecting other packets that have already started the pipeline is not trivial and entails arguably more overhead than the initial optimization saved. We leave it up to the programmer to decide whether she would like to optimize the program in view of this risk.

Ensuring Packet Correctness

So far, we disregarded the content of the misspeculated packet p_x. To deal with p_x we suggest sending it to the controller and ensuring correct processing there. The problem of split-brain processing of p_x needs to be further investigated, especially when the processing of p_x relies on state in the data plane.

Alternatively, p_x could be recirculated and each table augmented with an additional key indicating whether p_x has already matched the table in question. However, all metadata would have to be carried alongside p_x upon recirculation, since the respective tables would not be matched again and that metadata would remain uninitialized on the second run of the pipeline. This creates non-negligible overhead and questions the usefulness of the initial optimization, should packet correctness be required by the application. Please see Appendix A.2 for an in-depth analysis of this problem.

Example Program Structure

For illustration purposes, consider the detailed pseudo-code of Program 1 given in Code Snippet 2. For a full example written in P4, please refer to Appendix A.3.

Code Snippet 2 Actions and statements linked to `table_A`, `table_B`, `table_fix` and state modifier Y. In this scenario we can ensure state consistency, thanks to a state irrelevant dependency caused by field `srcAddr` (blue). Indeed, `table_fix` carries the state-relevant modification from `table_B` (orange).

```
table_A → action a_1 :
    modify(srcAddr);                              ▷ state irrelevant dependency
    modify(srcPort);
    modify(egress_port);
table_B → action b_1 :
    modify(srcAddr);                              ▷ state irrelevant dependency
    modify(dstAddr);
    modify(metadata.prevDstAddr);                 ▷ state relevant modification
table_fix → action fix_1 :
    modify(metadata.prevDstAddr);                 ▷ carried-over modification
...
if (metadata.prevDstAddr == 127.0.0.1)
    Y → register_write(myReg, dstPort);
```

There exists a write-after-write action dependency between actions `a_1` and `b_1` of tables `table_A` and `table_B` respectively. This dependency was removed through optimization constructs seen in the control flow in Program 1. However, `b_1` also performs a modification of metadata `prevDstAddr`, that is relevant for the state modifier Y. By carrying over this modification to action `fix_1` of table `table_fix`, which does not present a dependency with `table_A`, we can ensure that the modification is carried out, independently of a misspeculation. Therefore, state consistency can be guaranteed, even though the packet's content is not correct.

Chapter 4

Implementation

As a proof-of-concept, we implemented components of the classifier for the memory optimization (see 3.1) using python and symbolic execution engine z3 [16]. In this section, we present the overall system design, related toolchains and describe the inner workings of the classification mechanism.

4.1 Related Toolchains

Previous work [35, 45, 43] in symbolic execution for P4 follows the following workflow: a parser translates P4 code into an intermediate representation (IR), which is then used to generate code in a formal verification language. This code is then fed to a symbolic execution engine, such as K [41], or a theorem prover, such as z3 [16].

Alternative approaches [39] have translated P4 to C code or LLVM IR and then directly applied symbolic execution frameworks, such as KLEE [12] or crucible [20].

Unfortunately, we could not benefit from this work for multiple reasons. Firstly, only some implementations were made publicly accessible. Secondly, partial or missing documentation would have made code adoption difficult. Finally, previous work was written for $P4_{14}$ and it was questionable whether all necessary features needed for our classification were available.

4.2 Design

To perform the classification presented in Section 3.1 and verifying conditions for each offloadability type, we need access to the dynamic runtime configuration of the switch, namely installed rules. While we can rely on the compiler for static dependency analysis, our constraints go far beyond what is available to the compiler. Therefore, we resort to symbolic execution. More specifically, we use Satisfiability Modulo Theories (SMT) as a formal verification method.

4.2.1 System Overview

Figure 4.1 shows the general overview of the system. The core of our classifier is built around the SMT solver z3. Given a P4 program and associated rules, we generate a set of constraints that model the program. Using the constraint solver, we classify each rule.

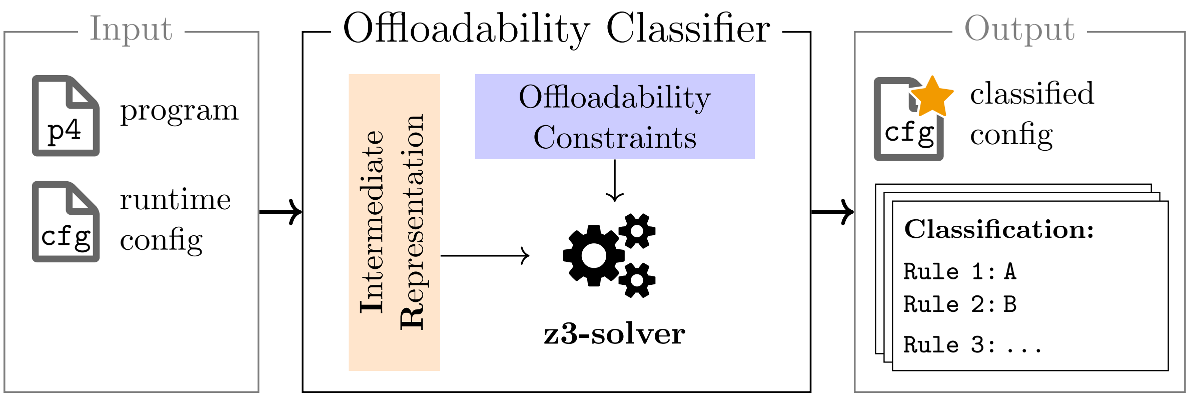

Figure 4.1: Given a P4 program and a runtime configuration, the offloadability classifier categorizes rules by their offloadability type. The symbolic execution engine is built around SMT solver z3.

4.2.2 SMT Basic Functionality

An SMT problem is a decision problem for logical formulas, which can either be `sat` (satisfied) or `unsat` (unsatisfied). The problem is formulated in so-called *constraints*, a list of equations to be satisfied. SMTs also support the use of basic data structures such as integers or arrays.

Given three integers x, y, z, a simple SMT problem could be given by :

$$\begin{aligned} 3x + 2y - z &\leq 4 \\ x &> 0 \\ y &\neq 0 \\ 0 < z &\leq 42 \end{aligned} \tag{4.1}$$

An SMT solver, such as z3, will then return `sat` and upon request an *interpretation*, which is a set of values that satisfy the problem: $x = 1, \quad y = -1, \quad z = 1$.

To use SMT, all P4 structures need to be translated into this form of constraints.

4.3 Building the SMT Translator

This section explains how we translate both P4 components and the offloadability conditions into SMT statements. To allow for different P4 flavors, we built an intermediate representation (IR) in python and use z3's python frontend to translate the IR into SMT-LIB [7] statements. We enrich our IR with a network analysis backend to represent the control flow of a packet, similarly to the control flow graph produced by the P4 compiler.

4.3.1 Translating P4 Components

A parser is required to automatically translate a P4 program of a given flavor ($P4_{14}$, $P4_{16}$, Tofino variants . . .) into our python IR. While we did not build such a parser, the data structures of our IR are built similar to P4 components and employ a modular structure. This makes building different parsers a trivial, albeit work-intensive, task. Our IR does not implement the whole specification of P4, we made abstraction of certain features when they were not necessary for the overall functionality of the classification algorithm.

Building an Intermediate Representation

The intermediate representation of P4 components is conceptually very close to the P4 components themselves. The main difference is that each component in the IR is directly linked to a directed acyclic graph (DAG), given by the control flow of the program. This allows each component direct access to its predecessors, successors, or internal components.

Code Snippet 3 Showcasing the modular structure of building IR components in python. In the background a directed acyclic graph between components is constructed automatically, giving access to an enriched version of the control flow graph, normally produced by the P4 compiler.

```python
drop_action = P4DropAction("drop_action")

set_nexthop_modifier = P4Statement(
    "set_nexthop_modifier",
    routing_metadata_nhop_ipv4,
    P4StaticAssignment(),
    "nhop_ipv4",
)
set_egress_modifier = P4Statement(
    "set_egress_modifier",
    standard_metadata_egress_port,
    P4StaticAssignment(),
    "port",
)
set_nhop_action = P4Action(
    "set_nhop_action",
    arguments=["nhop_ipv4", "port"],
    statements=[
        set_nexthop_modifier,
        set_egress_modifier,
    ],
)
ipv4_lpm_table = P4TableNode(
    "ipv4_lpm_table",
    keys=[ipv4_dst],
    actions=[set_nhop_action, drop_action],
)
```

Code snippet 3 shows an extract of the generation of P4 components using our IR. Our backend automatically generates a DAG (see Figure 4.2), which is then used to translate IR to SMT statements and generating constraints for each offloadability type.

Translating IR to SMT

In this section, we will describe how we translate our IR into SMT constraints. The result of each translation is a set of z3 constraints, whose union forms a model of the initial P4 program.

Variables. We translated P4 ”variables” such as headers and metadata as *BitVectors* in z3, to make sure we remain as close to the semantics of the P4 program as possible. BitVectors are a container of bits that support addition, subtraction, etc. through clauses that model digital circuits (*e.g.,* ripple-carry adders).

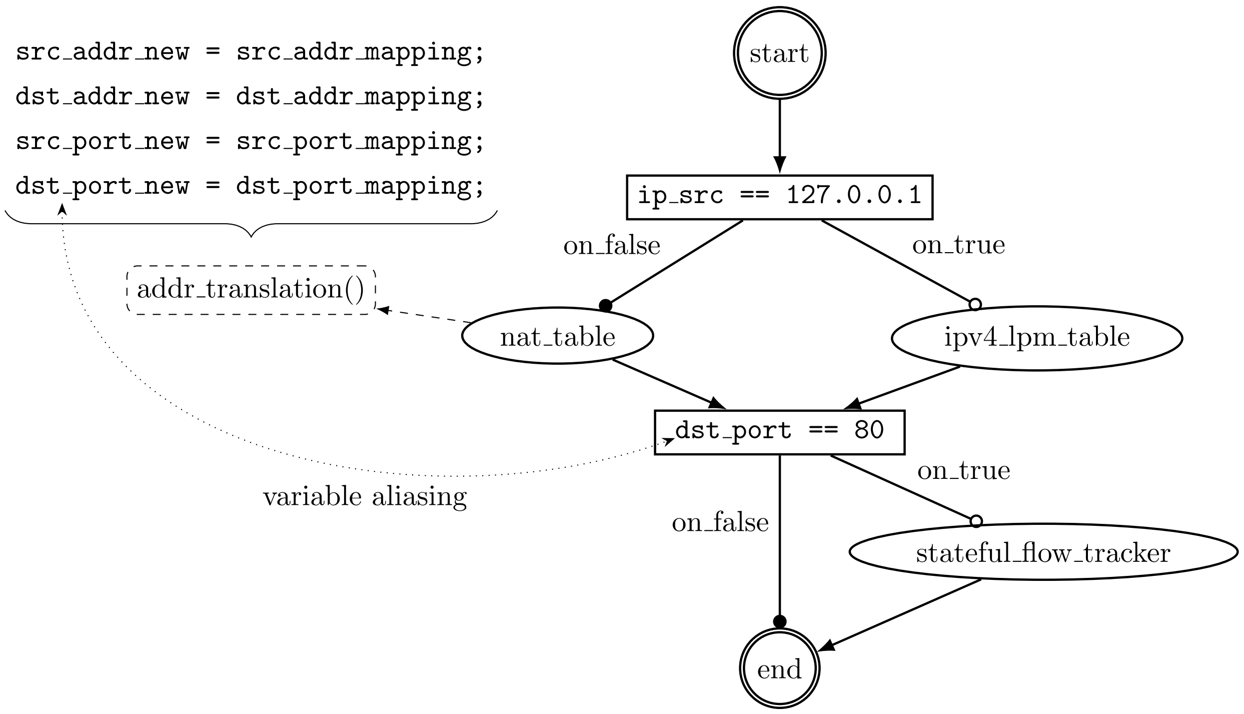

Figure 4.2: The graph analysis backend automatically generates a directed acyclic graph (DAG) from the intermediate representation (IR). The DAG is then traversed in topological order to translate P4 components to SMT constraints. Variable aliasing also uses the DAG to find the most "recently" used alias for a given variable.

Statement Translation. Note that z3 is generally *stateless*, which means that there is no form of order between statements. This means that a simple variable assignment in P4 such as

$$\texttt{ipv4.ttl = ipv4.ttl - 1} \tag{4.2}$$

cannot be simply translated into

$$ttl = ttl - 1 \tag{4.3}$$

because this would yield `unsat`. Indeed, there does not exist any value for *ttl* that can satisfy this equation. Instead, we need to resort to a form of *aliasing*, where we introduce new variables for each assignment:

$$ttl\_alias = ttl - 1 \tag{4.4}$$

Should `ipv4.ttl` then ever be reused in the code, we need to make sure that we select the right alias in the z3 model. Our IR is therefore enriched with a DAG that we can easily traverse to find the right alias for each variable (see Figure 4.2).

Table Translation. Translating P4 tables into z3 is fairly straightforward, because z3 supports constructs such as `Implies` (*i.e.,* $\implies$) or `IfThenElse`. Code Snippet 4 shows an extract of the z3 constraints generated for a table `b`. A table entry implies the execution of an action, which itself implies the execution of each one of the statements in that action. The table is then responsible for managing aliases and ensuring that the right alias is selected later on in the model, depending on which action was executed.

Registers. We modeled P4 registers using z3 arrays, which support storing and retrieval of values, given an index. Since we chose BitVectors as data structure over integers, we do not

Code Snippet 4 Showcasing an extract of the SMT constraints generated by our tool for a table `b` with one exact matching entry `entry1`, that triggers the execution of an action `action1`, which sets the variable `port` to `80`.

```
b_entry1 == And(src_addr == 14753),                    ▷ Entry exact matching on src_addr
b_action1 == Or(b_entry1),                 ▷ Action executed if any of its entries is matched
    ...
Implies(b_action1 == True,                           ▷ Action executes each of its statements
    port_b_action1 == 80),                      ▷ Generation of a new alias for variable port
Implies(b_action1 == True,                                 ▷ Actions are mutually exclusive
    And(b_action2 == False,
        b_action3 == False)),
    ...
Implies(b_action1 == True,                                  ▷ Alias management by the table
    port_b == port_b_action1),
Implies(Or(b_action1, b_action2) == False,       ▷ If no action executed, port unchanged
    port_b == port),
   ...
```

have to worry about out-of-bounds register accesses. Indeed, the wrap-around behavior of z3 is modeled after machine arithmetic of CPUs.

4.3.2 Translating Offloadability Types

We implemented a translation of offloadability types A, B and C. In the following, we describe how the conditions in Section 3.1.2 are translated into SMT constraints.

We implement an iterative algorithm: we gradually augment our model with constraints for the next offloadability type, if R_x is not of the previous offloadability type. First, we check if R_x is of type A, only then if it is of type B, and so on. This implicitly returns the overall offloadability type, as specified by the precedence relationship explained in 3.1.2.

Offloadability A is characterized by the non-reachability of state after the execution of a given rule R_x in table T. We model this behavior by requiring that there does not exist a packet p^*, that matches a rule R^* in a table T^* (successor of T), which triggers the execution of an action A^* that modifies state:

$$\begin{aligned} \nexists \quad p: \quad & T^* \text{ is successor of } T \\ & \wedge R^* \text{ rule of } T^* \\ & \wedge A^* \text{ action triggered by } R^* \qquad \Longleftrightarrow \qquad R_x \text{ is offloadability type A} \\ & \wedge M^* \text{ state modifier in } A^* \\ & \wedge p \text{ matches } R^* \\ & \wedge p \text{ matches } R_x \end{aligned} \tag{4.5}$$

Offloadability B means that there is reachable state S_x after T, but its access is independent of the existence of R_x. We model this by effectively duplicating the given program H and its constraints C into H' and C'. Then, we modify C' by removing the execution of the action A_x linked to R_x. If S_{tot} is the union of all states in the given program and S'_{tot} the union of all states in the duplicated model, then we require:

$$\forall \quad p: \quad S_{tot} = S'_{tot} \qquad \Longleftrightarrow \qquad R_x \text{ is offloadability type B} \tag{4.6}$$

Offloadability C means that there is reachable state S_x after T that depends on R_x, but it is read-only. This check is trivial because we simply add the constraint that the state access is not a state modifier.

Chapter 5

Evaluation

In this chapter, we evaluate the application of our framework on memory optimizations presented in Section 3.1 with respect to its effectiveness and the implementation presented in Chapter 4 with respect to its scalability.

All experiments are run on a machine with 8GB of RAM and an Intel i5-8250U 8 core CPU. The evaluation represents a qualitative analysis of our framework and proves the potential of our approach.

5.1 Effectiveness

To assess the potency of our classification, we statically analyzed the implementations of a wide range of P4 programs available online. For each program, we were interested whether there could exist table entries of different offloadability types.

Example	Offloadability Type		
	A	B	C
NetCache [26]	✓		
NetChain [25]	✓		
NetHCF [31]	✓		
LossRadar [33]	✓	✓	
PRECISION [8]	✓	✓	
Dataplane Routing [14]	✓	✓	✓
Speedlight [49]	✓		
P4-Guide [19]	✓		
SP-PIFO [2]	✓		
Blink [24]	✓		

Table 5.1: Offloadability types found in open-sourced P4 implementations of different networking features. Because of the single-feature nature of these implementations, we notice little to no existence of offloadability types B and C in some programs.

The quantitative question of *how many* rules of each type exist is not meaningful for our purpose, because one could artificially inflate the number of entries for a given table by producing appropriate traffic. This would directly impact the ratio of rules per offloadability type.

Furthermore, as most open-sourced P4 programs are proof-of-concept implementations of a certain feature, they would not be run individually in a production environment. As such, the examples present a code pattern, where the given feature is activated based on a "selector table": the feature is activated if there exists a rule in this table. In other words, if a packet matches a rule in this table, it will access state. This goes directly against our classification where we ask the inverse question to offload rules: Given a match on rule R_x, can we guarantee the nonexistence of state accesses (*i.e.,* offloadability type A)?

However, given several features in the same program, a potential mutual exclusivity of features would increase the number of offloadability types. Indeed, the combination of features would also greatly increase optimization opportunities in general. Thus, the assessment given here can be understood as a conservative estimation. We leave the merger of features and evaluation of their rule classification up for future work.

Table 5.1 shows which offloadability types can be found in different P4 programs. Since our implementation only supports classification of types A, B and C, we did not include offloadability type D. Note, that all programs contain offloadability type A, while offloadability type C was only found once. This is partly due to the nature of type C: when state is read, an index is used that was previously calculated independently of the type of state access (read/write). It is often also used for write access, which directly leads to a potential offloadability type D. Code Snippet 5 shows this structure found in many programs: the type of state access depends on the type of packet and not on the rule, so rules in `table_index` cannot be of offloadability type C. In view of those results, our analysis remains inconclusive as to the practicality of offloadability type C.

Code Snippet 5 Pseudo-code of a state access pattern found in many programs, leading to little offloadability type C rules. Indeed, the type of state access depends on the type of packet and not on the rule in `table_index`. Intuitively it makes sense that the same index is used for both read and write accesses, so that only one table is used for this purpose.

```
apply(table_index)                                ▷ sets index for register access
if(packet == SYN) {
    value = read_from_reg(register, index)
}
else {
     write_to_reg(register, index, value)
}
```

5.2 Scalability

Scaling our classification to bigger programs with more rules poses two fundamental problems.

First of all, bigger programs with more components take more time to be translated into SMT constraints. Secondly, symbolic execution commonly suffers from an effect called *path explosion.* Bigger programs lead to an exponential increase in the number of possible branches in the program code, leading to longer execution times. Applied on the classification of offloadability types, this translates to longer solving durations of the final SMT problem.

To tackle the first issue, we implemented a caching mechanism where the translation of IR components into SMT constraints is cached. This allows reuse of translations across different classifications of rules. Solving the second problem is quite more challenging. While z3 offers a variety of different strategies, each specialized in solving a certain kind of SMT problem, chaining them together is difficult and requires a deeper understanding of algorithmic proof methods.

Figure 5.1 shows the duration of rule classification of the program LossRadar for different types of rules. The figure reports maximum translation and solving times in milliseconds for 10 runs of classification for each rule, with and without the caching mechanism.

Focusing on the uncached results only, note how the overall classification time for rules $5-7$ is significantly higher than for other rules. Indeed, rules $5-7$ are of offloadability type B, while

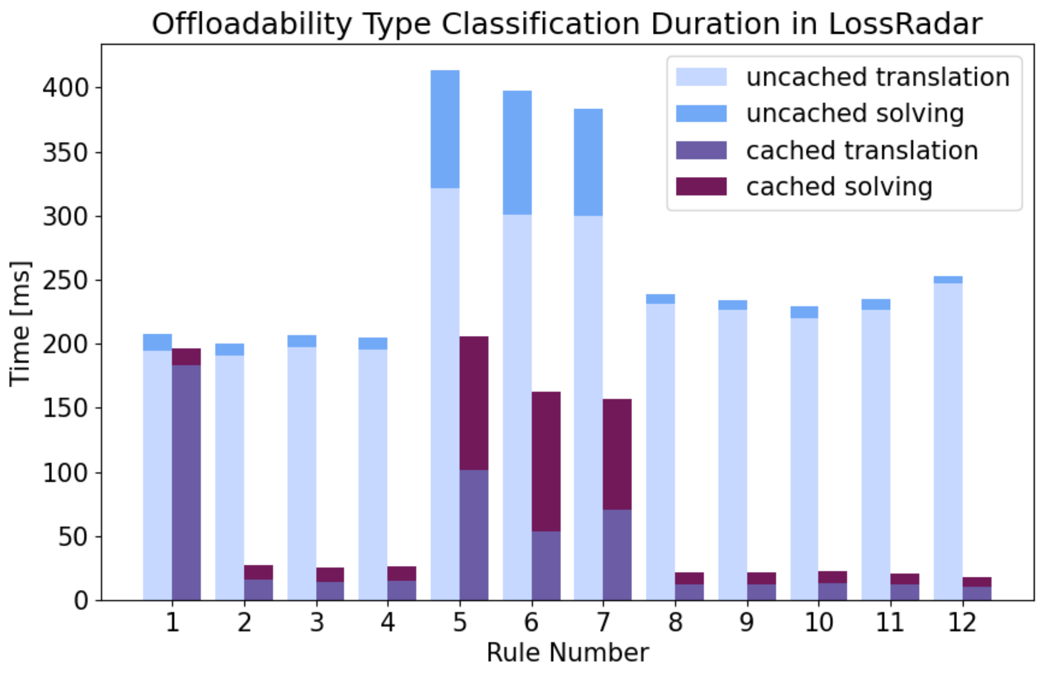

Figure 5.1: Maximum classification duration (over 10 runs) for rules of different offloadability types with and without a caching mechanism. Rules $5-7$ are of offloadability type B, resulting in longer classification and translation times. For all offloadability types, translation times are cut down significantly when using caching.

the others are of offloadability type A. This goes in line with the iterative algorithm presented in Section 4.3.2. Indeed, for offloadability type B, there is an internal duplication of the program (see Section 4.3.2) which explains both longer translation times and longer solving times of the SMT problem.

Comparing cached and uncached execution times, we notice two details. Firstly, the translation of rule 1 using caches takes as long as the method without caches. This is expected, because on the first run the cache is empty and needs to be filled. Indeed, subsequent runs take significantly less time. Secondly, the translation time is cut by factors of 12 (offloadability type A) and 3 (offloadability type B). This shows the potential of more advanced implementation-specific optimization techniques such as caching. Going one step further, one could make use of multi-core architectures by classifying rules in parallel, instead of classifying rules sequentially — currently only one core of the CPU is used for our classification. While this increases overall "throughput", we also see more potential for each individual classification. To reduce the solving time by z3, one could optimize the SMT strategies through novel learning approaches, such as FastSMT [5].

Chapter 6

Outlook

We dedicate the following sections to the current limitations of our framework and how we applied it to the memory and dependency optimizations. We also outline possible improvements and future research directions.

6.1 Limitations

Given the results in Chapter 5, we describe the limitations of the framework itself, of how we applied the framework in chapter 3 and of the implementation.

Applicability. Results have shown that despite an intuitive separation of offloadability types, little presence of offloadability type C questions its usefulness in practice. While the nature of the data set was not representative of production-ready P4 code, it remains questionable whether the separation between offloadability type B and C is appropriate. More experiments are also necessary with regards to offloadability type D. While it appears more complex than other types, we see huge potential in the disassembly of actions into statements.

Concerning the dependency mitigation, we focused on dependencies between actions. However, our detection and mitigation hook does not work for read-after-write matching dependencies. Detection mechanisms here are even more challenging because if a ”fixing” table were to match on the same keys as the optimized table, the removed dependency would be re-introduced.

Overhead. The cost of running the offloadability classification analysis is comparatively high to the usual latency (order of milliseconds) for adding a rule for the control plane. While more advanced implementation-specific optimization techniques can help, the fundamental problem of path explosion when using symbolic execution remains. However, the classification can be run in parallel with the rule insertion, while also parallelizing the classification for different rules.

For the mitigation of dependency optimization misspeculations, more memory resources would be needed to install detection and mitigation rules. This then raises the trade-off between resources. Is it worth having less dependencies but higher memory usage?

Model construction. While the framework lays out a structured approach to classifying optimizations, applying the concepts to concrete code transformations is not trivial. Balancing the complexity of the classification and the applicability range to cover a maximum amount of code structures is an open problem. Code sections can easily be marked as state modifiers; however, it is not clear how any given resource optimization interacts with each state modification. Concretely, the construction of models in Sections 3.1 and 3.2 is currently not easily portable to other resources.

6.2 Future Research Directions

This work challenges the prevailing definition of semantic equivalence on stateful switches and advocates a more refined view on packet processing operations. As such, we only scratched the surface of possibilities in an extremely vast space of semantically non-preserving optimizations.

First and foremost this raises a trade-off between packet correctness, state consistency and resource optimizations. Applied to the memory optimization: which rules should be offloaded, given the knowledge of offloadability types and their respective guarantees? Different applications have very diverse requirements for latency, packet correctness and state consistency. For example, it may be more desirable to allow miscounting in a bloom filter (*i.e., not* guaranteeing state consistency) used for rate limiting, rather than processing a certain type of traffic incorrectly (*i.e.,* ensuring packet correctness). Further research is required to translate these notions altogether into an efficient optimization problem.

Secondly, we focuse on a single special hardware architecture with specific constraints on packet processing, state modification and resource usage. As touched upon before, different resource optimizations require different mitigation and classification methods. This raises the question how our framework can be incorporated into new hardware architectures that offer different data structure models.

Finally, our model only focuses on a single device and a single optimization at a time. Past work in stateful network-wide abstractions (SNAP [4], Lyra [21], Flightplan [46]) has shown resource optimization potential utilizing a global view of the network. Exploiting our differentiation of semantic equivalence into packet correctness and state consistency, and applying these concepts on a network scale, even more aggressive and powerful optimizations can be unlocked.

Chapter 7

Summary

In this thesis, we explored the impact resource optimizations can have on the semantic correctness of P4 programs. To this end, we introduced new concepts to capture the packet processing behavior of P4 switches: packet correctness and state consistency. We then theoretically analyzed memory and dependency optimizations with respect to these concepts.

For memory optimizations, we proposed offloading rules to a remote entity and classified rules by their offloadability type, in order to capture their effect on stateful data structures and packet processing. For the dependency optimization, we introduced detection and mitigation mechanisms for misspeculated packets, under certain structural constraints.

Using a prototype implementation, we evaluated the effectiveness and scalability of our system and showed the feasibility of an automated classification engine. Our approach is a promising first step towards aggressive resource optimizations in the setting of partial semantic correctness.

Bibliography

[1] 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.

[2] Alcoz, A. G., Dietmüller, A., and Vanbever, L. Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 59–76.

[3] Applegate, D. A., Calinescu, G., Johnson, D. S., Karloff, H., Ligett, K., and Wang, J. Compressing rectilinear pictures and minimizing access control lists. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2007), SODA '07, Society for Industrial and Applied Mathematics, p. 1066–1075.

[4] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, Association for Computing Machinery, p. 29–43.

[5] Balunovic, M., Bielik, P., and Vechev, M. Learning to solve smt formulas. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 10337–10348.

[6] Barefoot. Barefoot tofino. `https://barefootnetworks.com/products/brief-tofino/`, accessed Sept 2021.

[7] Barrett, C., Fontaine, P., and Tinelli, C. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[8] Ben-Basat, R., Chen, X., Einziger, G., and Rottenstreich, O. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), pp. 313–323.

[9] Bennett, J. C. R., Partridge, C., and Shectman, N. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw. 7*, 6 (Dec. 1999), 789–798.

[10] 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.

[11] 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.

[12] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)* (San Diego, CA, Dec. 2008), USENIX Association.

[13] Cisco. Configuration guide for cisco ip sourceguard. `https://www.cisco.com/en/US/docs/switches/lan/catalyst3850/software/release/3.2_0_se/multibook/configuration_guide/b_consolidated_config_guide_3850_chapter_0110110.html`, accessed Sept 2021.

[14] Costa Molero, E., Vissicchio, S., and Vanbever, L. Hardware-Accelerated Network Control Planes . In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (2018), ACM.

[15] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, Association for Computing Machinery, p. 254–265.

[16] de Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, pp. 337–340.

[17] Dong, Q., Banerjee, S., Wang, J., Agrawal, D., and Shukla, A. Packet classifiers in ternary cams can be smaller. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2006), SIGMETRICS '06/Performance '06, Association for Computing Machinery, p. 311–322.

[18] Draves, R., King, C., Venkatachary, S., and Zill, B. Constructing optimal ip routing tables. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)* (1999), pp. 88–97 vol.1.

[19] Fingerhut, A. Demo 6. `https://github.com/jafingerhut/p4-guide/tree/5dc2d80c8282f5cb01a803c46e537704bf0f3715/demo6`, accessed Sept 2021.

[20] GaloisInc. crucible. `https://github.com/GaloisInc/crucible/`, accessed Sept 2021.

[21] Gao, J., Zhai, E., Liu, H. H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., and Yu, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 435–450.

[22] Govind, S., Govindarajan, R., and Kuri, J. Packet reordering in network processors. In *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–10.

[23] Group, P. A. W. P416 portable switch architecture(psa). `https://p4.org/p4-spec/docs/PSA.html`, accessed Sept 2021.

[24] Holterbach, T., Molero, E. C., Apostolaki, M., Dainotti, A., Vissicchio, S., and Vanbever, L. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 161–176.

[25] Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., and Stoica, I. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 35–49.

[26] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 121–136.

[27] 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.

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

[29] Karp, R. M. *Reducibility Among Combinatorial Problems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 219–241.

[30] Katta, N., Alipourfard, O., Rexford, J., and Walker, D. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2016), SOSR '16, Association for Computing Machinery.

[31] Li, G., Zhang, M., Liu, C., Kong, X., Chen, A., Gu, G., and Duan, H. Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)* (2019), pp. 1–12.

[32] Li, R., Zhao, B., Chen, R., and Zhao, J. Taming the wildcards: Towards dependency-free rule caching with freecache. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)* (2020), pp. 1–10.

[33] Li, Y., Miao, R., Kim, C., and Yu, M. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2016), CoNEXT '16, Association for Computing Machinery, p. 481–495.

[34] Liu, A. X., and Gouda, M. G. Complete redundancy detection in firewalls. In *Data and Applications Security XIX* (Berlin, Heidelberg, 2005), S. Jajodia and D. Wijesekera, Eds., Springer Berlin Heidelberg, pp. 193–206.

[35] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., and Foster, N. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 490–503.

[36] LLVM. Llvm's analysis and transform passes. `https://llvm.org/docs/Passes.html`, accessed Sept 2021.

[37] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[38] Meiners, C. R., Liu, A. X., and Torng, E. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. In *2007 IEEE International Conference on Network Protocols* (2007), pp. 266–275.

[39] Neves, M., Freire, L., Schaeffer-Filho, A., and Barcellos, M. Verification of p4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2018), CoNEXT '18, Association for Computing Machinery, p. 73–85.

[40] P4.org. Behavioral model version 2. `https://github.com/p4lang/behavioral-model`, accessed Sept 2021.

[41] Rosu, G. Specifying languages and verifying programs with k. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2013), pp. 28–31.

[42] Ruia, A., Casey, C. J., Saha, S., and Sprintson, A. Flowcache: A cache-based approach for improving sdn scalability. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2016), pp. 610–615.

[43] Schneider, T. Automatic generation of adversarial workload for programmable switches. `https://nsg.ee.ethz.ch/fileadmin/user_upload/TiborSchneider_AdversarialWorkload_Thesis.pdf`, 2019.

[44] Sonchack, J., Loehr, D., Rexford, J., and Walker, D. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (New York, NY, USA, 2021), SIGCOMM '21, Association for Computing Machinery, p. 731–747.

[45] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 518–532.

[46] Sultana, N., Sonchack, J., Giesen, H., Pedisich, I., Han, Z., Shyamkumar, N., Burad, S., DeHon, A., and Loo, B. T. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 571–592.

[47] Suri, S., Sandholm, T., and Warkhede, P. Compressing two-dimensional routing tables. *Algorithmica 35* (04 2003), 287–300.

[48] Wintermeyer, P., Apostolaki, M., Dietmüller, A., and Vanbever, L. P2go: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2020), HotNets '20, Association for Computing Machinery, p. 146–152.

[49] Yaseen, N., Sonchack, J., and Liu, V. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 402–416.

[50] 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.

[51] Zheng, P., Benson, T., and Hu, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2018), CoNEXT '18, Association for Computing Machinery, p. 98–111.

Appendix A

Appendix

For completeness purposes, we present additional insights and thought experiments in the appendix. Sections A.1 and A.2 show partially fruitful attempts at improving the classification and mitigations presented in Chapter 3. Section A.3 contains a full P4 example showcasing the mitigations presented in Section 3.2.

A.1 Exploration of Solutions to Guarantee Packet Correctness for Offloadability B

In this section we explore potential solutions to guarantee packet correctness in offloadability B. We refer to the terminology and notation introduced in Chapter 3. In short, packet correctness could be guaranteed under very specific assumptions, but we considered these cases marginal, which is why we did not pursue this research direction. The insight boils down to a reversal of wrongly modified packet headers in the control plane. However, because of possibly non-invertible transformations in the data plane, the control plane's actuation possibilities are very limited.

Offloadability B is characterized by an independence of the currently classified rule R_x and state accesses. However, since the packet p_x has to traverse the whole pipeline after a match on R_m and because of the miss in T, content in p_x might no longer be correct. For instance, there can be a read-after-write dependency between T and $Table_A$ with an action $Action_A$ that modifies a header of p_x. This modification will now be wrong because of the misspeculation (absence of R_x). If the mapping between rules and modifications (actions with their arguments) in $Table_A$ is not bijective or statements in actions of $Table_A$ not invertible, it will not be possible to reverse the wrong modification (due to a wrongful match in $Table_A$) in the controller before correctly applying T. Thus, we need to have access to the state of p_x right after the execution of A_m. However, cloning/generating digests in PSA "freezes" the state only at the end of ingress pipeline. We would need T to be at the end of the ingress/egress pipeline to make use of this feature. Should the latter condition be satisfied, we can guarantee packet correctness.

A.2 Exploration of Alternative Solutions for Dependency Misspeculation Mitigation using Recirculation

Early on, we explored the idea that recirculation of packets effectively increases the number of pipeline stages available to the programmer. Therefore, one might be able to use recirculation to mitigate misspeculations arising from dependency optimizations. However, this can potentially change the processing order of packets, which in turn can change the functionality of the algorithm. In the end we no longer pursued this direction, because of technical difficulties when trying to ensure state consistency.

We distinguish between the following scenarios:

No State. If there are no stateful data structures in the program and no packets are recirculated or cloned to the control plane, then reordering of packets does not alter the functionality of the algorithm. We can safely reorder packets as we wish.

Monitoring State. If there are stateful data structures in the algorithm, but they only have a monitoring role, then reordering should only have negligible impact. State purely used for monitoring purposes could be detected as follows. State is either only written to but not read from (we read it only from the controller for QoS monitoring or similar). Or state is also read from but only to notify the controller. This might be hard to detect in practice, but we can make some reasonable assumptions, such as only allowing cloning and sending the packet to the controller.

The impact of packet reordering can be described as follows. Let's say we want to keep track of average TCP window sizes and sample every tenth packet. If the order of the packets is changed because of recirculation, our average will be wrong. However, we can consider this impact negligeable. State is only used for monitoring and as such not crucial functionality of the code. Furthermore, such changes because of reorderings will not be noticeable in the sheer volume of correctly monitored packets.

Functional State. State is used for functionality of the program. We need to recirculate all packets whose state is altered by the recirculation of the misspeculated packet p_x. This needs to be done recursively, such that if packet p_1 is recirculated because of p_x and p_2 and p_1 have shared state, then p_2 needs to be recirculated too.

How can we detect shared state between (loosely) consecutive packets? Depending on the inter-packet arrival time, packets do not have to be consecutive to be "eligible" for recirculation, since recirculation takes a few cycles. So how do we know which packets have a dependency with respect to some data structure? We are given the following constraints:

- We need to know this before the state gets read/modified.
- We do not want to rely on a traffic profile.
- We cannot rely on the dependency graph, because it shows dependencies among data structures and not dependencies among packets.

The following example shows why we cannot rely on the dependency graph.

```
apply(table_green);   // --> modify_field(Y, 0);
apply(table_yellow);  // --> modify_field(X, 1); modify_field(Y, 1);
if (miss) {
    apply(table_special);  // --> read_field(X);
}
```

Notice, that there is no read-after-write dependency on `X`, as the reading and writing branches are mutually exclusive and can therefore be put in the same stage. After a P2Go dependency optimization, the code would look as follows:

```
apply(table_green);
if (miss) {            // added by P2Go because "unseen dependency"
    apply(table_yellow);
    if (miss) {
         apply(table_special);
    }
} else {      // account for misspeculation
    if (table_copy_yellow.hit()) {
        // we have a misspeculation and recirculate the packet
    }
}
```

Assume the following packets would arrive in this order to the switch: p_1 matching tables `table_green` and `table_yellow`, and then p_2 matching table `table_special`. Before the optimization this would result in:

```
p1:
    hit table_green -> Y = 1;
    hit table_yellow -> X = 1;
p2:
    miss table_green,
    miss table_yellow,
    hit table_special -> read(X) = 1;
```

After optimization and recirculation this would result in:

```
p1:
    hit table_green -> Y = 0;
    hit copy_table_yellow -> recirculated
p2:
    hit table_special -> read(X) = 0;
p1 after recirculation:
    hit table_yellow -> Y = 1;
```

Note, that the state read by p_2 differs in both versions. However, the dependency graph will not show a dependency between `table_yellow` and `table_special`. However, recirculating packet p_1 will make p_2 read a wrong value from X. The fundamental problem is that we don't know which state will be modified by a certain packet, before it actually reached that statement in the control flow. One would need sort of "canaries" to early-recirculate before critical state is read from/written to, similar to how stack canaries in C programs protect from stack overflows.

However, further exploration of this idea has shown inter-dependencies when using a form of P4 recirculation canaries resulting in more overhead than the initial dependency optimization saved.

A.3 Example Program Illustrating Dependency Optimization Mitigations

For completeness purposes, we show a full example program illustrating the dependency optimization mitigation. It contains ipv4 forwarding, udp access control and a form of Sourceguard [13] DHCP snooping. Please refer to the comments in the control flow section for the original and the optimized versions of the program.

```
#include <core.p4>
#if __TARGET_TOFINO__ == 2
#include <t2na.p4>
#else
#include <tna.p4>
#endif

#include "common/headers.p4"
#include "common/util.p4"

struct metadata_t {}

// ---------------------------------------------------------------------------
// Ingress parser
// ---------------------------------------------------------------------------
parser SwitchIngressParser(
        packet_in pkt,
        out header_t hdr,
        out metadata_t ig_md,
        out ingress_intrinsic_metadata_t ig_intr_md) {
    TofinoIngressParser() tofino_parser;

    state start {
        tofino_parser.apply(pkt, ig_intr_md);
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.ether_type) {
            ETHERTYPE_IPV4  : parse_ipv4;
            default : reject;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            0x11    : parse_udp;
            default : accept;
        }
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition select(hdr.udp.src_port) {
            0x44    : parse_dhcp;
            default : accept;
        }
    }

    state parse_dhcp {
        pkt.extract(hdr.dhcp);
        transition accept;
    }

}

// ---------------------------------------------------------------------------
// Ingress Deparser
// ---------------------------------------------------------------------------
control SwitchIngressDeparser(
        packet_out pkt,
        inout header_t hdr,
        in metadata_t ig_md,
        in ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md) {

    apply {
        pkt.emit(hdr);
    }
}

control SwitchIngress(
        inout header_t hdr,
        inout metadata_t ig_md,
        in ingress_intrinsic_metadata_t ig_intr_md,
        in ingress_intrinsic_metadata_from_parser_t ig_intr_prsr_md,
        inout ingress_intrinsic_metadata_for_deparser_t ig_intr_dprsr_md,
        inout ingress_intrinsic_metadata_for_tm_t ig_intr_tm_md) {

```

```
action discard() {
    ig_intr_dprsr_md.drop_ctl = 0x1;
}

action special_discard() {
    ig_intr_dprsr_md.drop_ctl = 0x1;
    hdr.dhcp.your_ip = 0x0;
}

action send(PortId_t port) {
    ig_intr_tm_md.ucast_egress_port = port;
}

action static_port() {
    ig_intr_tm_md.ucast_egress_port = 0x8;
    ig_intr_dprsr_md.drop_ctl = 0x0;
}

action l3_switch(mac_addr_t new_mac_da, mac_addr_t new_mac_sa, PortId_t port){
    hdr.ethernet.dst_addr = new_mac_da;
    hdr.ethernet.src_addr = new_mac_sa;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    ig_intr_tm_md.ucast_egress_port = port;
}

action fix_misspeculation() {
    // ported this statement from the original action of X
    hdr.dhcp.your_ip = 0x0;
}

action miss() {
}

table ipv4_host {
    key = {
        hdr.ipv4.dst_addr : exact;
    }
    actions = {
        send;
        l3_switch;
        static_port;
        discard;
    }
    size = 65536;
    default_action = static_port;
}

table udp_port_acl {
    key = {
        hdr.udp.dst_port : exact;
    }
    actions = {
        discard;
        miss;
    }
    default_action = miss;
    size = 65536;
}

table dhcp_offer_acl {
    key = {
        hdr.dhcp.your_ip : exact;
    }
    actions = {
        special_discard;
        discard;
    }
    size = 65536;
}

// TABLE COPY USED FOR MITIGATION
table dhcp_offer_acl_copy {
    key = {
        hdr.dhcp.your_ip : exact;
    }
    actions = {
        fix_misspeculation;
    }
    size = 65536;
}

DirectRegister<bit<32>>() test_reg_dir;
DirectRegisterAction<bit<32>, bit<32>>(test_reg_dir) test_reg_dir_action = {
void apply(inout bit<32> value, out bit<32> read_value){
    read_value = value;
    value = value + 1;
}
};

action register_action_dir() {
    hdr.dhcp.gw_ip = test_reg_dir_action.execute();
}

table reg_match_dir {
    key = {
        hdr.ethernet.src_addr : exact;
    }
    actions = {
        register_action_dir;
    }
    size = 1024;
    default_action = register_action_dir;
    registers = test_reg_dir;
}

apply {
    // OPTIMIZED WITH HOOK AND CONSISTENT STATE MITIGATION
    if (hdr.ipv4.isValid()) {
        if ((hdr.ipv4.ttl & 0xfe) != 0) {
            ipv4_host.apply();
            if (hdr.udp.isValid()) {
                switch(udp_port_acl.apply().action_run) {
                    miss : { if(hdr.dhcp.isValid()) { dhcp_offer_acl.apply();
    } }
                    default: { if(hdr.dhcp.isValid()) {
    dhcp_offer_acl_copy.apply(); } }
                }
            }
            if(hdr.dhcp.your_ip == 0x1928394) {
                reg_match_dir.apply();
            }
        }
    }
/*  ORIGINAL VERSION OF THE PROGRAM
    if (hdr.ipv4.isValid()) {
        if ((hdr.ipv4.ttl & 0xfe) != 0) {
            ipv4_host.apply();
            if (hdr.udp.isValid()) {
                udp_port_acl.apply();
                if(hdr.dhcp.isValid()) { dhcp_offer_acl.apply();  }
            }
            if(hdr.dhcp.your_ip == 0x1928394) {
                reg_match_dir.apply();
            }
        }
    }
*/
    // skip egress
    ig_intr_tm_md.bypass_egress = 1w1;
}

} // end of ingress

Pipeline(SwitchIngressParser(),
         SwitchIngress(),
         SwitchIngressDeparser(),
         EmptyEgressParser(),
         EmptyEgress(),
         EmptyEgressDeparser()) pipe;

Switch(pipe) main;
```