Synthesis of Network Switch Programs

Francisco Chan Carvalho Machado

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics

Supervisors: Prof. Fernando Manuel Valente Ramos
Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Fernando Manuel Valente Ramos
Member of the Committee: Prof. Maria Inês Camarate de Campos Lynce de Faria

June 2022

Acknowledgments

First and foremost, I want to thank my parents and brothers for their never ending encouragement and support. I'd like to express my gratitude to my girlfriend Barbara for always believing in me. To all of my family and friends, I couldn't do it without you, thank you from the bottom of my heart. I'd also like to thank Prof. Luís Pedrosa and Prof. Fernando Ramos for their guidance and supervision, as well as my colleagues João Tiago and Francisco Pereira for all of their help and advice.

Abstract

The emergence of programmable network hardware devices allowed engineers to deploy Network Functions that perform orders of magnitude faster than previous programmed software NFs. The improved performance of these devices comes at the expense of increasing programming complexity, which requires a thorough understanding of the underlying pipeline and hardware details. Some recent domain-specific languages, such as P4, have tried to optimize network packet forwarding by providing additional constructs and externs. However, the task of programming hardware devices still remains associated with steep learning curves and their integration in networks is limited to specialized engineers.

The state-of-the-art solutions address this challenge by means of rule-based approaches, often used in compiler design, to automatically generate code for these network devices. In this thesis we follow a different approach. We developed SyNAPSE, a program synthesis-based technique that leverages static analysis tools to generate NF implementations in P4 that are semantically equivalent to the original NF code. Our prototype reduces the effort required to program network devices by generating NF implementations for a P4 software switch and a standard x86 controller from NFs that were written in the well-known programming language C.

Keywords

Network Functions; Program Synthesis; Network Function Synthesis; Data Plane Programmability; P4;

Resumo

Na última década, foram introduzidos dois paradigmas que tiveram um grande impacto na indústria de redes: as Redes Definidas por Software (SDN) e a virtualização de Funções de Rede (NFV). Estes novos paradigmas introduziram um certo grau de programabilidade, aumentaram a flexibilidade e agilizaram a operação das redes. No plano de dados, em particular, o paradigma NFV permite correr funções de rede em software. As vantagens ao nível da flexibilidade trazem consigo o desafio do desempenho. Entretanto, a evolução nas arquiteturas de *chips* de redes resultaram em dispositivos de hardware programáveis com velocidades de processamento ordens de grandeza superiores às funções de rede virtualizadas. Em contrapartida, estes dispositivos estão associados a linguagens de programação de baixo-nível, e a sua introdução na infrastrutura exige por isso engenheiros de redes altamente especializados.

Como solução para este desafio, investigação recente propôs mecanismos de tradução baseados em regras, semelhantes aos usadas em compiladores, para este contexto. Neste tese propomos uma abordagem diferente: a utilização de técnicas de síntese de programas para sintetizar código para estes dispositivos de redes programáveis, com o intuito de reduzir a sua acentuada curva de aprendizagem. Com base nestas ideias, desenvolvemos o SyNAPSE, um protótipo que gera implementações de NFs para um switch P4 em software e para um controlador x86, a partir de NFs programadas em C. O protótipo aplica técnicas de síntese a partir de representações obtidas através de execução simbólica.

Palavras Chave

Funções de Rede; Síntese de Funções de Rede; Programação do plano de dados; P4;

Contents

List of Figures

List of Tables

Listings

Acronyms

ASICs	Application-Specific Integrated Circuit
CEGIS	Counterexample Guided Inductive Synthesis
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
FPGAs	Field-programmable Gate Array
LAN	Local Area Network
ML	Machine Learning
NAT	Network Address Translator
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
NPUs	Network Processing Units
NUMA	Non-Uniform Memory Access
P4	Programming Protocol-Independent Packet Processors
PISA	Protocol Independent Switch Architecture
PSA	Protocol Switch Architecture
SDN	Software Defined Networks
TCP	Transmission Control Protocol

VM	Virtual Machine
VPN	Virtual Private Network
WAN	Wide Area Network

1 Introduction

Contents

The latest advances in the networking industry have contributed to an era of unseen innovation and development. Similarly to the computing industry, the networking sector has traditionally been dependent on vendors to supply network products, or middleboxes, specifically designed to address specific problems other than traffic forwarding, such as security and performance. The substantial demand for these products paired with their inflexible deployment has resulted in massive costs to network operators. These constraints led to difficulties scaling networks as adding new middleboxes to already extremely complex network infrastructure required a lot of effort: complying with the established network policies, optimizations and quality-of-service requirements. These difficulties, which were more evident as networks grew in size, motivated paradigms that tried to tackle issues that were stifling innovation.

Software-Defined Networking (SDN) [1] was developed to address management issues by decoupling traffic forwarding (the data plane) from routing logic and network management (control plane). SDN also introduced network programmability by allowing the network control plane to be logically centralized on a software controller. SDN motivated the development of Network Function Virtualisation (NFV) [2]. NFV enabled developers to provide middlebox functionality previously performed by specialised network devices by using general-purpose commodity servers to run software Network Functions (NFs).

With the adoption of these new network paradigms, by decoupling the NFs from the hardware and the control plane from the data plane, the installation of fixed-function hardware middleboxes across the network was considerably reduced. This translated in a decrease of the overall network complexity and largely contributed to a more efficient and flexible deployment of NFs granting some degree of abstraction to program network devices.

SDN introduced a new approach to determine the forwarding behavior of packets processed by the hardware. Flow rules allowed for programmers to decide which packet-processing actions would be applied to packets that would match the designated match fields. In the early stages of SDN, the rigid switch designs only accepted matching a fixed set header fields. However, the available packet-processing actions and the header fields have progressively increased to more general hardware designs. Nowadays, data plane packet processing can be programmable in hardware, and still guarantee line rate processing. This data plane programmability has emerged as another alternative to the inflexible middleboxes, allowing NFs to be programmed and compiled to programmable networking hardware capable of running at Terabit speeds.

1.1 Motivation

Unfortunately, programming network devices and their multi-stage forwarding pipelines can involve a lot of effort and time. Programs have to consider all the available resources and the forwarding pipeline limitations in order to run at line-rate [3]. These devices are usually associated with low-level chip-specific languages, with inconvenient learning curves. For example, optimizing linear-rate programs requires a deep knowledge of the underlying hardware constraints to take full advantage of all their capabilities. Some domain-specific language (DSL) languages, such as the P4 language [4], offers a way to program these switches, allowing to specify how packets are processed in the data plane. P4 programs are designed to be target-independent as they can be compiled against a wide range of forwarding elements, including hardware or software switches, network interface cards, and programmable network appliances. Despite enabling data plane programmability, P4 does not make it easy the task of programming the device. Programming in this language requires specialized knowledge of the target architecture, in order to guarantee line rate processing of packets.

When expanding and scaling networks, developers are therefore faced with the decision of whether to use software NFs, which provide greater flexibility and familiarity, or programmable hardware, which offers improved packet-processing speeds but is more difficult to program and optimize according to network constraints. If a network developer chose the latter, building a network function would take far more time and effort than implementing the NF in software using a more familiar programming language. Additionally, developers must consider the effort that programmable network hardware will require to optimize whenever network alterations are required.

In recent years, some tools have shown success in generating network-wide *configurations* utilizing program synthesis techniques (i.e., to program the *control* plane). Following on from these ideas, in this thesis we suggest to address the complexity of programming these intricate hardware devices by also following a program synthesis approach, but for the network *data plane*. Contrary to the state-of-the-art approach [5] that uses rule-based techniques, a program-synthesis solution can consider a larger search space of solution, potentially enabling the generation of packet-processing code that is faster and better optimized to the considered architecture. However, using program synthesis techniques instead of traditional compiler techniques has one main disadvantage: finding programs that fit the specified hardware constraints without sacrificing performance can lead to increased compile time, as this process is a complex combinations problem and the search space can grow exponentially with the program size. The challenge is therefore to develop domain-specific heuristics that prune and optimize the search space.

1.2 Contributions

This thesis adds to the state-of-the-art by proposing SyNAPSE. SyNAPSE is a tool that is able to generate new implementations of NFs that were originally written in C (using the Vigor [6] libraries), into semantically equivalent NFs that offload packet-processing to a P4-programmable software switch and to a x86 SDN controller.

The prototype incorporates two previously existing tools, Vigor and Maestro, which perform static analysis of software NFs. It receives as input a network function programmed in C language in the context of Vigor – a software-verification framework. The Vigor framework explores the use of symbolic analysis to check NF behavior against a specifications for correctness. These static analysis methods are then used by Maestro to build a sound and complete representation of the NF behaviour. Maestro is used to automatically parallelize sequential implementations of NFs, by distributing traffic across CPU cores while preserving the semantics of the sequential implementation. We adopt the model of the NF behaviour of Maestro to obtain an intermediate representation of the NF, from which we are able to generate code for both the P4 switch and the SDN controller, resulting in a semantically equivalent NF to the original NF.

Our prototype allows network developers to write NFs in a language they are more familiarized with, thus reducing the time and effort required to deploy NFs directly in P4.

1.3 Organization

The remainder of this dissertation is divided into four chapters.

In the following chapter, we analyze the state-of-the-art in Software-Defined Networks, Network Function Virtualization, and Data Plane Programmability. We will investigate the motivation for program synthesis and some existing frameworks and tools. Additionally, we present the two tools on which this prototype builds on, namely Vigor and Maestro. Then, Chapter 3 explains the design and implementation of the SyNAPSE prototype, and how it generates new implementations of NFs. This prototype can be described as a sequential set of three modules that continuously modify the initial input until the generated code is obtained. In Chapter 4, we evaluate the prototype, and in chapter 5 we conclude and discuss future work.

2

Related work

Contents

In this chapter, we start by presenting some backgorund information on Network Function Virtualization and Software Defined Networks. We then introduce program synthesis and verification, and present state-of-the-art solutions that apply these techniques in the context of computer networks.

2.1 Software-Defined Networking

Computer networks are complex systems of interconnected nodes that can be arranged in a variety of topologies and subjected to many requirements. In the past years, cloud datacenters and service providers have seen increasing complexity in their network infrastructures and policies. The number of network elements in infrastructure ramped up with the emergence of dedicated hardware to implement specific network functions, such as firewalls, network address translators (NAT) and intrusion detection systems (IDS). To comply with these standards, datacenter objectives include optimizing performance while coping with continuous network link failures, all while maintaining security and quality-of-service requirements [7]. As a result, network implementation and subsequent maintenance have become more difficult, requiring highly qualified and specialized engineers, resulting in high operational expenses.

SDN emerged as a solution to this overwhelming problem, enabling efficient network configuration and providing network programmability by allowing the network control plane to be logically centralized on a software controller. By decoupling traffic forwarding (the data plane) from routing logic and network management, SDN technology seeks to simplify network design, implementation, and management (control plane). The data plane is responsible for processing all incoming packets according to basic rules. A controller establishes these rules depending on network architecture and global network policy goals. These rules are encoded in forwarding tables populated by the control plane and usually processed at very high requirements [8].

In SDN, control logic is moved to an external entity called the SDN controller, a logically centralized system to manage and control all resources available represented by one or multiple controller instances. Three main elements can be distinguished (Figure 2.1): the controller, and its southbound and northbound APIs. Controllers serve as an intermediate component between the network control applications and the data plane. Applications can program the network by defining a set of functions and services through a northbound API, which subsequently interacts with underlying network services using the southbound API. The controller also collects network information, such as state of the resources or changes in topology, to continuously adapt network configurations according to desired policies.

In SDN, forwarding decisions are flow-based instead of destination-based [1]. Flows are an abstraction that group traffic by matching a set of packet fields to a set of actions, unifying behaviour of multiple network devices, like switches, routers, and other middleboxes. This abstraction increases flexibility for network programmers. This concept was introduced by OpenFlow [9] with *flow rules* that defined forwarding behaviours and an interface for controlling the data plane. A flow rule comprises match fields,

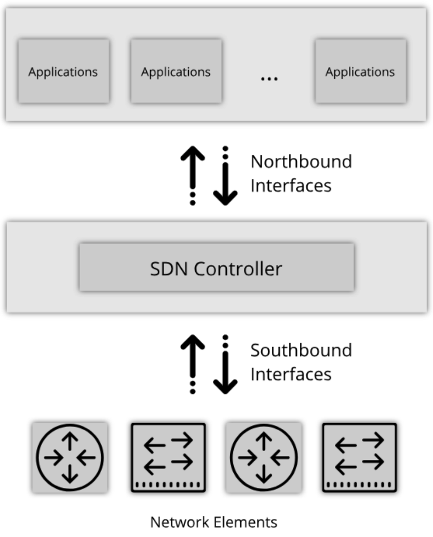


Figure 2.1: Simplified view of an SDN architecture.

counters and set of instructions. Packets are compared against specific patterns and if a match occurs packets will follow through with the configured action. These actions can include for example, dropping, forwarding to one or multiple ports, changing the packet headers and sending the packet to the controller for further processing. In the latter case, the controller can subsequently generate new flow rules and send them to the switch. Each network element in the network maintains a Flow Table with all the controller defined rules. In OpenFlow 1.1, the Flow table abstraction was refined with the adoption of chaining multiple flow tables, where flow entries point to another tables flow entry.

Another defining feature was the OpenFlow protocol which allowed the secure transmission of flow rules from the controller to the switch. The first implementations of SDNs didn't get much attention from the network community, however, the introduction of OpenFlow in 2008 gained significant traction and resulted in the creation of the Open Networking Foundation (ONF) in 2011, to promote SDN and OpenFlow. Since then SDN has seen wide adoption and multiple APIs and Controllers have been developed.

2.1.1 Data Plane Programmability

OpenFlow introduced new abstractions and the possibility of programming the network control plane but the technology was limited to match previously determined header fields and to chip-specific capabilities. In the following years, new switching chip architectures contributed to the increased flexibility while maintaining processing speeds required for deployment in networks. For example, the Reconfigurable Match Table [3] (RMT) model allowed the forwarding plane to be changed in the field without modifying the hardware, and the Protocol Independent Switch Architecture (PISA), an architecture that allows

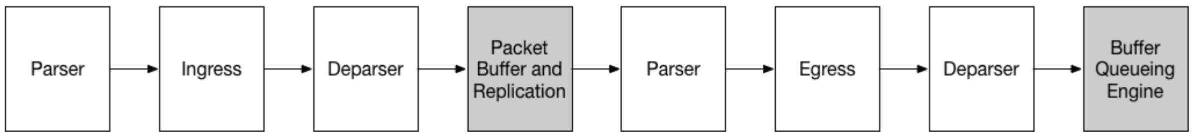


Figure 2.2: Programmable blocks of the Portable Switch Architecture.

forwarding pipelines to be programmed by high-level languages, such as P4 [4]. The possibility of programming functionality onto data plane resulted in a turning point for how ASICs (Application-Specific Integrated Circuit), and network devices overall, were developed.

The general approach to implement new functionalities onto fixed-function ASICs, followed a top-down design: New functionality had to be first standardized according to clients needs. Then the new design would have to be implemented in ASICs by chip designers. The complete process was slow hurting flexibility and hindering innovation [8]. The emergence of programmable network data planes allowed network engineers to dictate exactly how each packet should be processed. Directly programming the forwarding pipeline to implement new functionalities became a task operators could perform without depending on vendors, leading to rapid design cycles and quick innovation.

2.1.2 P4 Language

The P4 programming language was presented in 2014 [4] as a proposal for how OpenFlow "should evolve in the future". A revision to the P4 language was released in 2016, resulting in the P4$_{16}$ language specification [10]. P4 is a domain-specific language (DSL) to program forwarding planes. This DSL is protocol-independent, as it allows developers to specify the structures of the packet headers and how they are extracted and processed, and target independent as it supports multiple hardware architectures. P4 uses a compiler to map target hardware resources to the target-independent description, generating the necessary configurations [4].

P4 programs specify the behaviour of the programmable blocks of a given architecture. For example, the Portable Switch Architecture (PSA) Model [11] is composed of two pipelines: ingress pipeline (composed of a parser, ingress, deparser and a Packet buffer and Replication Engine (PRE)) and an egress pipeline (composed of a parser, egress, and Buffer Queuing Engine (BQE)). Packets traverse the pipeline in the order specified in 2.2. The P4 programmable parser enables user defined packet headers according to implementation needs. The main logic of the P4 program is defined in the ingress/egress blocks. In these blocks, the programmer can implement custom actions and define match-action tables with different combinations of matching entries, matching types and associated actions. Finally, the user can use these actions and/or tables freely, for example, depending on the parsed packet header elements. Besides defining packet processing throughout the pipeline, the architecture also defines a set of P4 externs that extend P4 functionality, for example, counters, meters and registers which allow to maintain state between packets.

Similar to OpenFlow 1.1, pipelines are composed of multiple match+action stages, except in P4 they can be processed in parallel. To address dependencies between headers Table Dependency Graphs (TDGs) are used. This process is performed by the P4 compiler to identify possible conflicts before mapping match+action tables to their location in the pipeline. After being validated in the parser, packet headers are passed to an ingress match-action pipeline determining operations to be applied to each packet. The ingress deparser identifies which packet contents and metadata are sent to the buffer. Buffers are not part of the P4 architecture and are used to place packets waiting for their respective actions, reducing congestion. Metadata fields can carry information that are not contained in packet headers but are relevant to execution, for example, timestamps used for some traffic shaping policy. Both metadata and packet headers can be used as match fields in match+action tables and are processed in equal manner. Packets can be forwarded to one or to multiple egress ports (multicast), dropped or placed in queues where traffic shaping policies are applied. In the multicast case, the packet is replicated the necessary number of times before being sent to the PRE, where they are placed in a buffer and processed when possible. The PSA also allows for packets to be recirculated or resubmitted. In this case, they are sent to the beginning of the pipeline. PSA also defines a library of commonly used constructs, called *externs*. *Externs* allow P4 code to be extended with new constructs not supported by the language but made available by the target platform.

To summarize, SDN enhanced flexibility enabling a logically centralized control plane to manage network devices from different vendors and with diverse characteristics. Data plane programmability enabled new functionality to be implemented faster and more efficient usage of hardware resources. This trade-off between programmability and hardware performance made SDN a trending technology since it first appeared. SDN has seen wide adoption across data centers, Wide Area Networks (WANs) and access networks and has complemented, together with P4, the development of technologies such as Intent-Based Networking and In-Band Network Telemetry.

2.2 Network Function Virtualization

As mentioned before, the number of network elements in cloud datacenters and service provider networks have increased in order to accommodate more users and services. Users have specific constraints and requirements for their traffic related to security, isolation and quality-of-service, which result in complex policies. This functionality is commonly implemented as Network Functions (NFs) and is often supported by dedicated intermediary hardware, also known as ”middleboxes,” which increase forwarding efficiency by decoupling specific network functions from the forwarding plane.

To comprehend the impact of the network architecture covered in this section, it's important to understand the market background of networking industry prior to it. Network functions were mostly implemented by proprietary equipment - vertically integrated solutions sold by large companies that developed all underlying technology, from hardware to middlebox operating systems. Thus, a network could demand the deployment of multiple middleboxes in order to offer a given service.

As networks grew in size and complexity, more functionalities were necessary to comply with user requirements and the increasing data line rates. Implementing a new functionality and updating equipment required network administrators to continuously purchase and store new integrated solutions directly to vendors. Scalability was also an issue for large networks, which would require as many middleboxes as forwarding elements [12], and the exponential difficulties in integrating solutions from different vendors in the same network. Combined with the slow-paced approach of producing new equipment mentioned in Section 2.1.1, this model of developing networks where implementing new functions depended on available solutions hindered innovation. All these constraints lead to high Operating Expenses (OPEX) and Capital Expenses (CAPEX) [2]. Telecommunications Service Providers (TSPs) started noticing these increasing costs and were forced to find more dynamic approaches to building and maintaining networks.

Network Function Virtualization (NFV) has been proposed to address the above mentioned challenges offering a new design to deploy services and improving resource management. NFV aims to decouple network functions from the underlying equipment offering more agility to deploy services and better resource management. This technique eliminates the network dependency on specialized hardware to perform network functions. Instead, network functions can be treated as plain software and decomposed in sets of Virtual Network Functions (VNFs) that can be deployed in standard physical servers. Such convenience makes scalability of networks possible and the deployment of services much faster, a task which once required network operators to purchase new equipment and develop appropriate skills to manage and integrate it. With NFV, capital expenses were also reduced as network functions could be deployed in commodity servers.

The concept of NFV was first suggested in October 2012 in a white paper [9]. The paper was the result from collaborative work from leading telecommunications companies calling for industrial and research action. The premise of this work was to decouple functions of proprietary hardware and move them to virtual servers, deployed in a wide range of hardware appliances. High-performance computing could be offloaded to virtual servers, for example, to COTS (Commercial off The Shelf) clusters, benefiting from affordable prices due to economies of scale. Some of the telecommunications companies that worked in the white paper selected ETSI (European Telecommunication Standards Institute) as the responsible entity for the NFV Industry Specification Group (ESTI ISG NFV)

The ETSI NFV framework consists of three main components: Virtualized Network Function (VNF),

VNF Infrastructure (VNFI) and VNF Management and Orchestration (VNF-MANO) [2].

1. NFVI corresponds to the environment in which VNFs are deployed. It includes every network element that provides either processing, storage, or connectivity to VNFs, thus including both hardware and software resources. Virtual resources are abstractions from the underlying physical resources, obtained using a virtualization layer.
2. VNF is the implementation of network functions deployed on virtual resources. VNFs can be deployed over multiple VMs depending on their internal components.
3. NFV MANO is responsible for managing all underlying resources and corresponding virtualised instances, performing operations such as configuring VNFs and any underlying infrastructure.

Although the Network Function Virtualization (NFV) architecture borrows concepts initially proposed in SDN to address challenges imposed by static architectures, no dependency exists between them. While both pretend to implement automation and virtualization, their goal is distinct. NFV main focus is on decoupling NF from specialized hardware whereas SDN separates the network control from packet handling. Nevertheless, leveraging both the network virtualization and logically centralized intelligence - *SDN-based NFV* - can lead to great value as VNFI benefits from having a central orchestration point.

Software-defined NFV can be applied to service-chaining and service deployment replacing time-consuming complex tasks of installing, connecting hardware to implement new functions or validating and testing network infrastructures. This approach allows maximizing the usage of network resources and reducing the overall costs of providing/adding services.

2.2.1 Architectures

The virtualization of NFs and the advances in data plane programmability (Section 2.1.1) have resulted in a wide-range of devices used for general forwarding and performing network functions [8], from general-purpose hardware to hardware designs like ASICs and FPGAs (Field-programmable Gate Array).

Network Processing Units (NPUs) are specialized accelerators employed in switches and NICs (Network Interface Cards). They target network packet processing being capable of performing network-specific operations such as load-balancing and encryption. NPUs are optimized to perform parallel computation to process packets from individual network flows and have powerful dedicated memories such as SRAMs (Static Random Memory Access), DRAMs and TCAMs. These memories are organized in a specific manner to aid packet-processing by hosting frequently accessed data and forwarding tables used to determine packet forwarding.

FPGAs are integrated circuits based around a matrix of configurable logic blocks and can be reconfigured in the field as per client needs. FPGAs are typically programmed with hardware description languages, like Verilog, but can also, in some cases, be programmed in C or P4.

Additionally, these devices offer high performance, being primarily used to offload packet processing from servers.

ASICs , contrary to FPGAs, have more rigid designs, implementing fixed pipelines or processing steps. As a result, they can process packets significantly faster than any alternative, but their programmability is limited. As seen in Section 2.1.1 reconfigurable ASIC architectures have been proposed, for example, RMT [3] and PSA [11] enabling programmability to implement custom network functions and forwarding, with the same performance of their fixed-function counterparts.

General-purpose Servers have been deployed at massive scales in datacenters to implement network processing. Traditional NIC interfaces are not optimized for high-performance packet processing, as they are designed to be used in simpler contexts.

In server-based solutions, processing packets requires copying packet's data from NIC buffers to the CPU, where modifications are performed before copying data back to another buffer. This simple series of actions is difficult to implement in general-purpose hardware.

To accelerate network processing in commodity hardware some frameworks have been developed to increase processing rates, including DPDK [13] and Netmap [14]. For example, *zero-copy* [15] uses buffers visible both to applications and the NIC, avoiding costly memory copy operations. *Kernel bypass* gives direct NIC control to userspace processes to avoid system calls and context switching. These techniques avoid copying from kernel buffers and system calls, saving CPU cycles and memory bandwidth. When a packet is transmitted over a network, a sequence of operations must be followed, including the acquisition of locks for the respective queue. Processing packets in batches (groups) reduces the number of necessary lock acquisitions and minimizes system call overhead by reducing the number of PCI-E bus registers.

The integration of SDN and NFV in real networks contributed for increased flexibility allowing companies to easily deploy new services over a broader range of hardware appliances. This translated into an increasing ability to scale the network according to its needs. The minimal dependency of middleboxes impacts greatly their cost. Besides these benefits, running software-based implementations of network functions in commodity hardware lead to the lack of reliability in these software NFs. Frameworks that perform software verification to NFs, such as Vigor, have been developed with the goal of mitigating possible security vulnerabilities and other potential bugs.

2.3 Vigor as an Analysis Framework

Vigor [6] is a software stack and toolchain for building and running software network middleboxes with the ability to automatically verify their correctness, while preserving competitive performance and developer productivity. Vigor allows NF developers to write NF code in C on top of the DPDK I/O framework. It uses libVig, a library of formally verified data structures, to store persistent states across packets. Symbolic execution is performed by the KLEE engine. This technique is used for live code path extraction which is then used to generate Call Paths. Instead of real values, symbolic execution analyses the program simulating the execution with symbolic values. These values are shaped in order to reflect operations and for any given branches presented in the program, the symbolic execution continues the simulation on both paths independently.

Besides verifying NF code correctness, the Vigor verification process includes verification of the entire NF stack to guarantee correctness in its entirety — libraries, packet-processing frameworks, operating system and device drivers. Verifying all these components would be inefficient and lead to path explosion given the complete stack dimensions. As a solution, its authors built a small, custom operating system (NFOS) that can be symbolically executed in an exhaustive manner, and modified both DPDK (in order to make it verifiable) and KLEE - to discard unnecessary paths and to speedup the process [16].

The authors guarantee that NFs are semantically correct, crash-free and memory safe [6] in four steps:

1. Obtain all live paths through the code - call paths - using exhaustive symbolic execution.
2. Convert symbolic execution traces to programs, where each one of them represents a path through the stateless code.
3. Annotate each trace with lemmas corresponding to the NF specification
4. Validate each annotated trace, using the pre- and post-conditions from the libVig API contracts.

Vigor uses symbolic execution to analyse the NF code and generates call paths. Each call path represents a possible execution path and collectively, the call paths generated by Vigor represent the complete behaviour of an NF and show all possible paths through the code triggered by a packet's arrival.

In this thesis, we propose the use of the call paths utilized for NF verification as the initial input to the SyNAPSE prototype. To these complete and sound representation of the NFs behaviour we will later be apply synthesis-based techniques to generate NF implementations.

2.4 Program Synthesis

Program synthesis is the task of automatically finding programs that satisfy given high-level formal specifications [17]. The process of finding correct programs constitutes a complex search problem that can be optimized by using different techniques, like complementing the specification with a general structure of the solution (Section 2.4.2) or examples of the program input and respective outputs (Section 2.4.1). The search for correct programs is a hard combinatorial problem because the *search space* of candidate programs grows exponentially with program size, which translates into increased compile time - the main drawback of a program synthesis approach.

Synthesizers are characterized by three key dimensions: the kind of constraints that represent user intent, the search space of candidate programs, and the search technique it employs. User intent can appear in forms of logical specification, examples (Section 2.4.1), traces or partial programs (Section 2.4.2). Logical specifications are difficult to express as they declare logical relations for a program's inputs and outputs. A trace is a detailed step-by-step description of programs behaviour according to its inputs, showing how they are transformed.

Because the *search space* should capture a high number of candidate programs whilst maintaining search efficiency, some optimization approaches are capable of pruning it efficiently. We decided to highlight the following approaches:

- *Divide-and-conquer*, a *Deductive search* strategy that recursively reduces the search problem into sub-problems. Analyzing smaller parts of the program individually can improve synthesis performance.
- *Constraint solving*, strategy that usually involves two main steps: *constraint generation*, where the applications are able to translate synthesis problems into constraints; and *constraint resolution*, where the constraints generated on the previous phase are solved.

2.4.1 Programming by Example

Many Program Synthesis implementations explore programming by example (PBE) techniques to infer correct program implementations. In PBE, the specifications given as input to the synthesizer consist in input-output examples. More generally, examples demonstrate behaviours/actions the system should have for different input states. Using examples specifications makes PBE more tractable than other program synthesis approach and are much easier for users to specify [18]. There are three main components in a PBE engine.

- The Search algorithm that efficiently finds programs that mimic behaviours specified in examples. The *search space* is pruned using divide-and-conquer techniques.
- Ranking Techniques that rank correct programs among the options that satisfy examples.
- The User Interaction Model, to guide synthesis and avoid selecting wrong programs.

The authors in [18] make an interesting comparison between Machine Learning (ML) and PBE as motivation to suggest that ML techniques are able to enhance the performance of PBE engines. Both involve prediction on new data after example-based training, yet the way they get to these predictions is very distinct. While ML requires huge quantities of training to generate sophisticated models with almost perfect precision for new inputs, PBE is expected to learn using few examples and generate solutions that provide perfect precision on new inputs. In any component, PBE engines use heuristic techniques, meaning that they can't guarantee that the search method or approach to the problem is optimal. ML can be used to learn and improve such heuristics, for example by analysing the structure of input-output examples and capturing critical features.

Ranking techniques can also benefit from machine learning. Ranking schemes are used to differentiate programs that only perform as expected for given examples, from correct programs by giving them higher rankings. The subtle differences between both programs can be identified with the intent of building efficient learning rankers. User interaction models can accelerate synthesis process, for example, the user can rank multiple synthesized programs based on preference. In this module, PBE engine comes across multiple heuristic decisions including which programs to present to the user and in what order. The correctness of these decisions can be improved with ML.

2.4.2 Sketch

Sketching [19] allows programmers to design algorithms using *sketches* and high-level insights, leaving implementation of low-level details to be synthesized automatically. In Sketching, the programmer will provide a sketch and define the expected behaviour of the program, either in a set of tests the generated code must pass or in the form of reference implementation. A sketch is a partial program describing a general solution where difficult statements are unspecified — *holes*. The correct value for all *holes* are derived from this high-level strategy, using inductive synthesis, leaving the implementation of low-level details to be solved automatically by synthesis procedures.

Sketch relies on the *Counterexample Guided Inductive Synthesis (CEGIS)* algorithm to determine correct values for *holes*. CEGIS combines a SAT-based inductive synthesizer and an automated validation procedure, that checks if a candidate implementation is correct and produces new counterexamples. CEGIS alternates between two phases: (1) the *synthesis phase*, where an inductive synthesizer generates candidate programs that assign values to all holes in an attempt to satisfy the input implementation; and (2) the *verification phase*, where an automated validation checks if generated candidate program is indeed correct. In the cases the validation isn't accepted, the validation procedure produces counterexamples that demonstrate the failure, which can are then passed back to the synthesizer to inform its next iteration. The idea behind CEGIS is to utilise a checking tool capable of generating counterexamples to avoid testing the implementation on inputs that will fail in identical ways. By exposing problems associated to each candidate program, the synthesizer will gradually produce solutions closer to the desired solution and effectively guide the synthesis process.

Sketching often requires that developers are knowledgeable of the various low-level languages which is not suitable for practical programming.

Holes are marked with generators defining a set of expressions that can be selected by the synthesizer for their completion. The generator used to define unknown values is represented by '??'.

The repeat statement can be used in cases where the programmer has doubts about the number of times assignments should be in a loop. The construct *repeat(n)* produces *n* copies of the statement and solves each copy independently. Sketch synthesis can be applied to multiple complex problems [19]. It successfully implemented an AES cipher by synthesizing fragments of code, for instance, the reference implementation used was transcribed from the NIST standard. The performance of the generated code by the sketch was found to be 10% less efficient when compared to hand written optimized code. Given the complexity of the problem, the quality of the optimized code was considered a good result. Many implementations of program synthesis combine techniques to generate low-level code for networking devices, developing new synthesis procedures for the different platforms. We explore these next.

2.5 Network Synthesis

Configuring data plane devices requires hardware expertise in the existing programmable network substrates. The ability to abstract the underlying hardware is one of the obstacles we face in this thesis: how to program a P4 programmable data plane from a generic network function specification, considering the underlying pipeline architecture and constraints.

Programmable devices and their respective programming languages have benefited from remarkable evolution in recent years, but are still at an early age and face challenges related to lack of abstractions for hardware features and capabilities, inhibiting their deployment in large scale networks. In this section, we will study how current synthesizers use program synthesis to generate different types of implementations. Systems will be separated in three different types according to their synthesis objective:

1. Configurations Synthesis, where systems are able to generate configuration that implement network wide policies (e.g, Cisco and Juniper) in conventional networks.
2. SDN Synthesis: systems able to generate policies to be executed on top of a SDN controller.
3. NF Synthesis: systems able to generate programs that implement network functions.
4. Synthesis of network programs: where systems are able to generate low-level configurations for network switching devices.

2.5.1 Configuration Synthesis

Network re-structuring can be highly disruptive to live traffic as the process of modifying and adjusting configurations for individual nodes is demanding and error-prone. In this section we analyse tools that aim to solve this issue by synthesizing optimized router configurations that implement network-wide policies in conventional networks, avoiding operator misconfigurations and any subsequent network outages. In fig. 2.3 we demonstrate a general example for these kind of synthesis tools.

Inspired by recent work on program synthesis, Beckket *et al.* [20] developed **Propane**/**AT**, a system capable of generating BGP configurations from high-level specifications for network topology, routing policies and fault-tolerance requirements. This framework was designed to be a improved version of

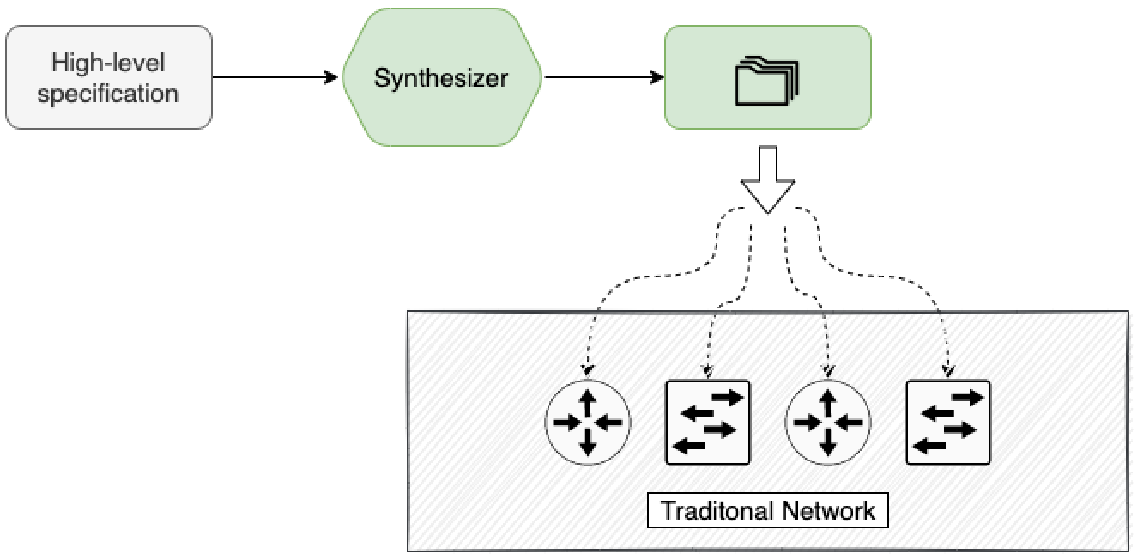


Figure 2.3: Illustrative example of Configuration Synthesis tools.

Propane [21], which introduced program analysis and compilation algorithms to translate regular policies into graph-based intermediate representations from which BGP configurations could be derived. These tools used regular expression to represent the various network specifications.

Additionally, Propane/AT takes as input the policy for the abstract topology. The policy consists of routing policy and a fault-tolerance policy. The synthesis process is divided into three phases. First, the routing policy and abstract topology are combined into a product graph (PG) containing the routing information for the topology in a policy-compliant manner. The algorithms used to manipulate Graphs are adapted from Propane. Second, Propane/AT checks compliance with the fault-tolerance policy. If met, a template for each node is generated in a third step. Transforming the template into BGP configurations — *concretization* — is done using a vendor-independent BGP configuration language. Results show that this framework is effective in real-world topologies and policies and configuration synthesis is considerably faster than systems that operate over concrete topologies.

At the time, the current existing tools supported a limited number of protocols, for example, Propane/AT was limited to BGP configurations and Genesis was limited to generating static router configurations. Addressing this limitation, El-Hassany *et al.* proposed a system capable of extending its supported routing protocols. **SyNET** [22] was able to synthesize correct network configurations for multiple and interacting routing protocols (OSPF and BGP) and static routes from requirements specified by an operator. Differently from the previous tools, SyNET does not uses regular expressions to describe network policies but Stratified Datalog, an expressive and complex declarative programming language suitable for describing routing protocols and capturing behaviour of the network in a clear and declarative manner, and a allowing operators to model routing protocols, the interactions between them, the topology of the network, and requirements such as reachability, path requirements, and traffic isolation. To synthesize configurations, SyNET applies an iterative algorithm to generate configurations by reducing input synthesis problem to SMT constraints. SyNET scales to real-world networks, synthesizing configurations for networks with 32 or less routers within an hour. For larger networks, with more than 64 routers, SyNET can take more than 24 hours to synthesize configurations, leaving some space for optimization. Another relevant topic discussed in SyNET is the idea that it is possible to take advantage of networks that are organized hierarchically around some regions and whose configurations can be synthesized independently. Although SyNAPSE does not consider network topologies, this idea could be applied to the binary decision diagram our Search Engine takes as input. SyNET approaches synthesis in a distinct manner from the above mentioned synthesis-based compilers by using a declarative logic language to describe routing protocols. The authors compared SyNET to tools of the likes of Propane/AT and Genesis and affirmed that SyNET is a more general system, as it allows for routing protocols to be added in stratified Datalog to synthesize configurations.

El-Hassany *et al.* presented a tool with the same goal of assisting network operators in modifying existing network-wide configurations to comply with new routing policies, using Sketch (section 2.4.2). Similarly to SyNET, **NetComplete** [23] supports static routes, OSPF, BGP and additionally, other routing requirements such as load balancing and traffic isolation are supported. The synthesizer scales much better for large networks, performing orders of magnitude faster than NetComplete. The synthesizer takes three inputs: the network topology, routing requirements and configuration sketches. NetComplete follows three steps to synthesize configurations for network-wide configurations. Firstly, the static routes defined in the routing requirements are synthesized. After that, BGP router-level configurations are synthesized from BGP requirements and policy sketches, computing a propagation graph with the order of BGP announcements. This process can also produce a set of constraints that need to be enforced in the OSPF synthesizer. Finally, the OSPF synthesizer uses CEGIS to comply with all requirements. This is known to be a complex search problem mainly for larger networks. CEGIS significantly improves OSPF synthesis compared to naive synthesis, where all requirements are simply introduced in a constraint solver to discover a model that computes link costs. With this search method, NetComplete evaluation results showed that the system is at least 600 times faster than SyNET and, as mentioned before, it is capable of synthesizing configurations for large topologies for which SyNET times out.

2.5.2 SDN Synthesis

With the emergence of Software Defined Networks, network controllers allow operators to implement a wide range of policies controlling switches in a network through SDN protocols. In this section we mention tools that generate configurations that are executed on top of a SDN from high-level specifications

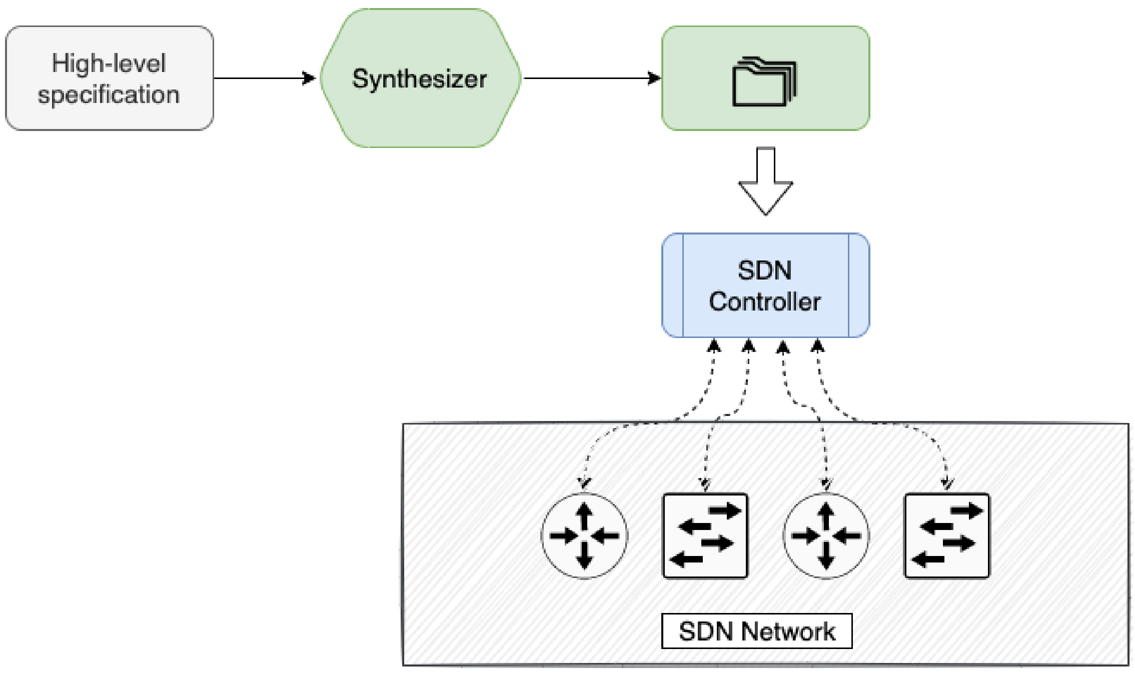


Figure 2.4: Illustrative example of SDN Synthesis tools.

of network policies and changes in the network state, as illustrated in fig. 2.4. The high-level domain-specific languages developed in the past years have greatly reduced the complexity of programming controller interfaces, however, programming SDN networks is still limited to a very reduced number of specialized network developers.

Towards the mitigation of this condition, Saha *et al.* [24] developed **NetGen**, a tool that automates the process of synthesizing configurations for the changes in the data-plane, and a specification language to express desired changes in an existing network. NetGen updates configurations by finding the minimal amount of changes to the network to satisfy a high-level specification.

NetGen presents a high-level language, based on regular expressions, to express desired changes in an existing network. The synthesis algorithm is capable of finding new network changes based on abstractions and constraint-solvers that generate new data plane states, different from the current implemented configurations in the network. The algorithm uses a state-machine to find the most reasonable number of changes to obtain the new data plane state. A distinct aspect of NetGen is that paths can be synthesized independently without affecting existing synthesized paths. NetGen was tested for larger networks with success with the authors proving its scalability by using NetGen in real-world networks of large dimensions. In comparison with previously developed synthesis-based compilers (2015), NetGen was more flexible and more widely applicable. It also approached configuration synthesis from a different perspective, as existing synthesis-based compilers like FatTire, Alloy and Merlin attempted to synthesize configurations for specific requirements, whereas NetGen uniquely synthesized new configurations by performing changes to existing ones. The use of regular expressions by tools allowed developers to define desired traffic patterns, however, Subramanian *et al.* [7] verified that the use of regular expressions can lead to increased number of constraints introduced in the SMT solver, thus slowing down the search process. Subramanian *et al.* presented **Genesis**, a network management system for multi-tenant datacenter networks and the use of *tactics*, a restricted form of regular expressions, that blacklists paths in a network based on patterns that are not desired. In Genesis, rich policies can be specified in a declarative manner allowing SMT constraint solvers to be used to program the underlying data plane, more specifically, to synthesize switch forwarding tables, that are subsequently introduced in network devices by a controller. By allowing developers to use isolating policies, a feature which was not implemented in similar previous tools, **Genesis** sped up significantly the synthesis procedure. Genesis is also capable of handling failures using a mechanism called *minimal repair* that calculates the minimal number of switches whose rule tables are modified before transitioning from a disrupted data plane to a new, and once again, policy-compliant state. To synthesize policies authors use a divide-and-conquer strategy to partition synthesis problems into smaller ones. These tools were significant advances in the area of program synthesis for network-wide configurations, however they were somewhat limited as Genesis only supported static routes and Propane/AT only supported BGP configurations.

Yuan *et al.* followed a different approach to generate SDN programs, presenting a scenario-based programming tool named **NetEgg** [25]. NetEgg allows network operators to program policies by describing representative example behaviours, from which it synthesizes automatically policy implementations to be executed on top of a centralized controller. The scenarios NetEgg receives as input are composed of examples of packet traces and their corresponding actions to each packet. The tool is capable of synthesizing automatically controller programs that are consistent with the example behaviours while the interpreter executes the policies according to incoming network events on the controller and infers rules that can be pushed onto switches [25].

NetEgg provides a configuration language to express network behaviour in representative scenarios, where packet variables and fields are typed. A packet-type consists of a list of names of fields of the packet along with their types. NetEgg also provides a library with standard packet fields and actions, such as flood, send(port), and modify(f,*v*) (modifying the value of a given field). Before the synthesizer tries to generate a policy according to these scenarios, it checks for any existing conflicts between them. If any exist, they are displayed to the network operator to be resolved.NetEgg can generate policy implementations within seconds with less overhead, when compared to other POX [26] implementations (the platform used for SDN control applications). The interpreter was tested for its Rule installation process, and results show that rules are installed correctly from the first incoming packet-in event.

NetEgg pioneered the approach of using examples to write network programs, however it only considered networks where the control plane was centralized in a single controller. In an SDN network, the control plane can be physically centralized or distributed, with controller processes running on multiple network nodes. **McClurg** *et al.* [27] addresses the problem of programming distributed controllers by taking a buggy decentralized program and inserting the necessary synchronization to make it correct. The authors highlight two concurrency challenges specific to networks with multiple controllers. They are *controller races* and *packet races* and both capable of introducing serious problems such as packet loss. The packet race happens when one of the controllers removes a forwarding rule, to introduce a new one. In between the two events the controller receives a packet, resulting in a dropped packet. The controller races happens when only one of the controllers adds rules that forward incoming packets in an incorrect manner. Its authors use a program synthesis approach to simplify writing distributed controller programs, so that programmers don't need to focus on keeping track of possible interleavings of controller processes and inserting low-level synchronization constructs which constrain interactions among controller processes to ensure the specification is always in compliance.

McClurg uses a CEGIS algorithm to find synchronization constructs to correctly synchronize the event net. CEGIS uses an *event net repair engine* to synthesize candidate event nets from the input event nets and a finite set of known counterexample traces and uses an *event net verifier* to check whether the candidate satisfies the LTL property, producing a counterexample trace if not. The developed prototype is deployed efficiently in several network topologies.

2.5.3 Network Function Synthesis

Yanjun Wang *et al.* [28] suggests an approach to learn the objectives of a network design through iterative interactions with the network architect. Frameworks mentioned earlier are capable of generating a solution assuming well-defined objectives, but in practice it can be challenging for an architect to precisely state the desired objectives. Its authors argue that besides the challenge of solving well-defined problems, the task of obtaining a formal specification that reflects the architects goals is also a complex problem. The challenge in network design relates to the determination of objective functions that reconciles multiple criteria, such as resource allocation and traffic engineering goals (latency, throughput, fairness). This approach is motivated by PBE systems (section 2.4.1), however, PBE techniques don't suit *comparative synthesis* as it requires the architect to specify and provide feedback on exact objectives. In this context if the architect could provide such feedback, he would have written the precise objective function. To explain the synthesis process we will consider the example mentioned by the authors, where the goal is to synthesize an objective function on both throughput and latency of a network design. The synthesis process starts with one or more sketches (section 2.4.2) or templates of typical objective functions, that reconcile throughput and latency. Initially, random weights are assigned to each unknown variable and the system queries the user about the ranking of a given set of concrete scenarios. User's preferences are then recorded in a Direct Acyclic Graphs (DAG) used to discard incorrect solutions and provided better scenarios, *i.e.* with better SMT-generated candidates, from which the user will continue to choose. As the user provides feedback in each iteration, the DAG grows in size, the SMT solver produces better candidates, and scenarios start to converge to the preferred user scenario. Eventually, if the preference graph is rich enough the SMT will return unsatisfiable, meaning only one possible solution is viable. Because the initial inputs are randomly assigned the experiments are non-deterministic. The authors conclude that (1) the developed prototype is capable of synthesizing correct object functions from multiple interactions with the user, and (2) the most optimal solution is for the user to rank 3 scenarios at time, which only reduced the amount of interaction moderately but decreased the total time of synthesis significantly. Similarly to this implementation, our tool should be able to discard solutions and make decisions to achieve a desired metric by manipulating DAGs, for example, ”reconcile throughput and latency”.

2.5.4 Network Program Synthesis

In this section we will examine compilers that use program synthesis to program configurations onto programmable network devices. Although these devices allow to deploy NFs onto networks and support packet processing at high speeds, programming them its a task that requires knowledge of the pipeline and other hardware details. Even with the recent development of DSLs targeting these devices, there is still a significantly learning curve for the average operator. Program synthesis can address this issue and has been used automatically generate packet-processing code that can result in faster processing/-forwarding. In fig. 2.5 we demonstrate a general description for network program synthesis tools.

Haoxian *et al.* developed **Facon** [29], a tool that addressed the proliferation of DSLs for programmable networks enabling automatic generation of *control plane* programs using arbitrary DSLs, based on input/output examples. Instead of generating device configurations, like NetEgg does, Facon generates the program whose function is to generate data plane configurations. The synthesizer takes two kinds of inputs, either the mentioned examples or Network DataLog (NDLog) relation schemas, and produces an NDLog program that satisfies all input-output examples.

The developed prototype was only used to evaluate feasibility and performance of Facon, thus only 4 simple programs were synthesized: a learning switch, a recursive reachability program, a stateful firewall, and a program for application-based forwarding. All were synthesized within a reasonable time budget, even for programs with a larger input size (a stateful firewall and application-based forwarding), as Facon is able to reduce the number of iterations to search within huge search spaces. Additionally, because Facon generates a program instead of configurations, the programs can be moved between networks with different underlying architectures. Generating the programs from input/output examples also allows Facon to transform from one DSL to others. Facon proposes search algorithms and heuristics, to guide the search direction, that can serve as inspiration to improvements to our first prototype. However, they target control plane programs, while we target data plane programs. Our challenge is therefore different as the data plane programs are more low-level, closer to the hardware architecture.

One of the challenging characteristics of programming hardware devices relates to their *all-or-nothing* nature, where if a program can fit within pipeline resources, it will run at line rate; otherwise, it will not run at all. Usually this task is handed over to a compiler, but standard rule-based compilers often are unable to rewrite programs so that they fit within the switch limited resources. Motivated by these drawbacks and by the achievements of program synthesis in related areas, Gao *et al.* proposed synthesizing code for hardware devices, presenting **Chipmunk** [30, 31] as a compiler that uses the SKETCH synthesizer to transform high-level programs (P4 or Domino) to switch machine code that fits within resource limits.

The synthesizer takes two inputs, a specification and a *sketch*, and generates a pipeline implementation. The sketch (section 2.4.2) is a partial problem with holes that represent unknown values in a finite range of integers that describe low-level machine code.To speed up synthesis, its authors developed a technique called *slicing* where each slice corresponds to a simpler synthesis problem. After synthesized, each slice is merged by stacking the resulting pipeline on top of each other composing the final hardware implementation. Other optimizations where introduced in the Chipmunk compiler, such as scaling up synthesis to larger inputs, constant synthesis and canonicalization. Chipmunk was designed with the intention to be *retargetable*, i.e to apply the same underlying program synthesis to multiple backend targets, generating machine code for different pipeline architectures by analysing declarative specification of given targets. In order to accomplish retargetability, the authors developed a DSL — the pipeline description language — to specify hardware pipeline capabilities. A switch hardware simulator was used to compare Chipmunk to Domino, in order to evaluate its performance against rule-based compilers. Domino [32] is a high-level language to program data-plane algorithms. Domino programs are compiled into low-level code that can run in programmable line-rate switching chips. Some noticeable findings included:

- Chipmunk is able to compile programs that are rejected by Domino, yet Domino's compile times are faster than Chipmunk.
- Chipmunk generates code with fewer pipeline stages — a highly constrained switch resource
- The slicing technique speeds up synthesis process by 51 times, on average.

The approach used here to build a retargetable compiler, as well as the optimization techniques used, are also highly relevant to this thesis and have informed our design choices. However, they generate a compiler from P4 to machine code, while our goal is to generate a P4 program from a high-level language program.

Other approach with a similar goal to Chipmunk – and therefore different to ours – is **Lyra** [33], a high-level language and compiler that enables programmers to efficiently program the data plane through a *"one-big-pipeline"* abstraction that allows programmers to use simple statements to express their intents without worrying about the underlying hardware details. Lyra also supports simultaneous deployment of multiple programs and programs that need to be executed in distributed ways, challenges mentioned in 2.5. The motivation example presented in Lyra's paper is a *load-balancer* that requires that NF to be deployed in different nodes according to their functionality in the network.

Data plane programs are written in chip-specific languages (P4 and NPL) that are tightly coupled to chip-specific architectures. Nevertheless, Lyra is capable of synthesizing code pieces for different chip-specific architectures while meeting functional correctness specified by the input Lyra program. Its key methodology consists in encoding all logic and constraints into an SMT problem and use an SMT solver to find an optimal implementation and deployment strategy for a target network.

The single pipeline abstraction is composed of chained modules that can be divided into Front-End and Back-End, taking as input high-level Lyra program, an algorithm scope describing each algorithm's placement, and a network topology and its configurations.

Because the constraints introduced by the P4 and NPL programming languages are distinct, each DSL has its own synthesis algorithm for generating chip-specific code.

To optimize and improve efficiency of resource usage, Lyra tries to reduce the number of P4 generated tables related to metadata fields that are redundant to program functionality. This mechanism can reduce by up to 50% the number of generated P4 tables. Other optimizing techniques consist of specifying requirements, for example, maximizing the usage of one particular switch, or minimizing the overall usage of switches. In Lyra's evaluation, the challenges mentioned in 2.5 are surpassed as the compiler is able to program distributed network functions and generate chip-specific code for multi-vendor switches. Furthermore, the code produced by Lyra uses fewer hardware resources than human-written programs.

To the best of our knowledge, Zhang *et al.*'s **Gallium** [5] is the closest work to ours. Its authors start from our main premise: with programmable hardware, developers can obtain performance gains significantly superior to what network functions deployed in software middleboxes can offer. However, as we also argue, the procedure of offloading NFs requires that operators manually select the components and write the P4 code. Zhang *et al.* [5] suggested **Gallium** to achieve this offloading automatically, transforming an input software network function into a functionally equivalent middlebox implementation that has two parts - a P4 program that runs on a programmable switch and a C++ program that runs on a traditional middlebox server. The intent of generating these two parts is to split the functionality between them and to improve performance by offloading packet-processing to a programmable switch. This goal is the same as ours. However, the strategy they use to synthesize the programs is different.

Gallium follows a rule-based approach: it splits the input middlebox into three partitions that are traversed by incoming packets in the following order: pre-processing partition, non-offloaded partition,

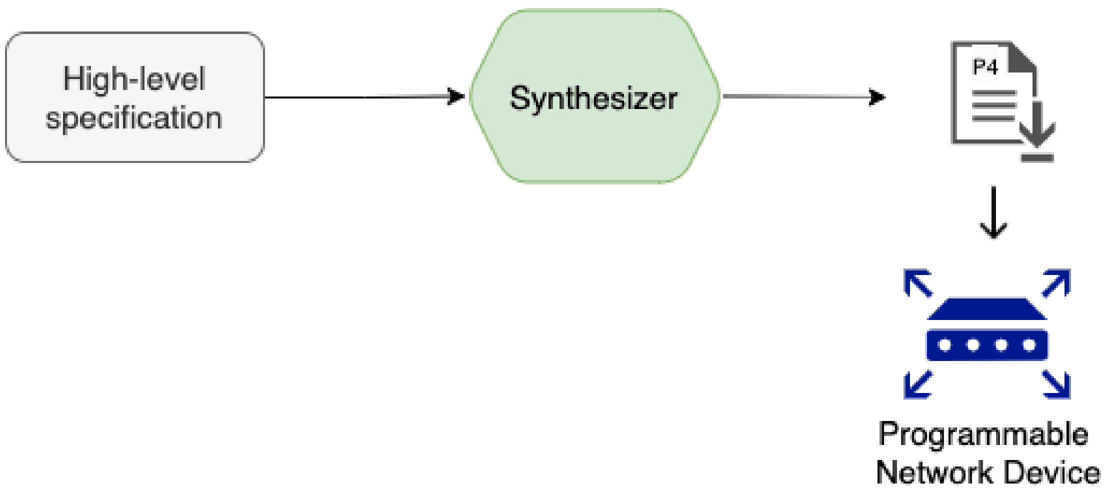


Figure 2.5: Illustrative example of Network Program Synthesis tools.

and post-processing partition. Pre-processing and post-processing are run on the programmable switch as a single P4 program, while the non-offloaded is executed on the middlebox server. Gallium must ensure that this order is followed, as well as the functional-equivalence between the source code and the generated program. Gallium keeps the instruction-level dependencies equivalent, and guarantees that the generated P4 program satisfies any resource constraints of the programmable switches. For each partition and according to its code, a control flow graph (CFG) is built. The generation of P4 code is done by mapping CFG instructions to P4 primitives, for example, temporary variables in CFGs correspond to metadata fields in P4 and Maps correspond to P4 Tables.

The main limitation of a rule-based approach such as Gallium is that its problem space is very limited. Our rationale in SyNAPSE is that by following a program synthesis-approach we can search a larger problem space to obtain better code translations.

2.6 Summary

In this section we studied how current synthesizers use program synthesis techniques to solve challenges introduced by data plane programmability and network paradigms, like SDN and NFV. These approaches reduce the risk related to error-prone tasks performed by network developers, saving time and effort as well as generating better solutions. The tools covered in this section are summarized in table 2.1. These program synthesis tools are organized by the technologies and methodologies used to synthesize configurations for both traditional and SDN networks, as well as complex network hardware devices.

For conventional networks (section 2.5.1), we examined 3 tools: **Propane/AT** and **SyNet** [22] that employed constraint-solving techniques, and **NetComplete** [23] that generated configurations for conventional networks using sketching and CEGIS. These tools simplify the process of individually configuring network devices to comply with continuous link failures and changing network policies. Following these tools, in the SDN synthesis section 2.5.2, we studied tools that solved similar challenges in SDN networks: **NetGen**, **NetEgg**, **Genesis** and **McClurg**. These four tools are capable of generating rules to be executed on top of a SDN controller, in order to comply with new policy implementations.

Network program synthesizers are the frameworks that are more relevant to our work as they are able to synthesize configurations for the programmable network device that require developers to be acquainted with their respective low-level languages. **Facon** focuses on generating SDN control programs. In contrast, **Chipmunk** [31] and **Lyra** [33] focus on generating efficient data plane machine code (starting from either P4 [31] or a new language [33]). Differently, the goal of **Gallium** and **SyNAPSE** is to start from a NF written in a high-level language, and both have the goal of offloading packet-processing to a programmable network device to improve the NF performance. However, **Gallium** follows a rule-based compilation approach, while we use program synthesis techniques. The benefit of a program synthesis approach is that we use techniques (from the Vigor toolchain) that result in a complete representation of the original NF program (the Call Paths). This representation increases the likelihood of finding more and better NF implementations in our search space that satisfy the original constraints. Additionally, Vigor NFs can be programmed in C, a widely popular programming language that is not limited to network developers.

Paper	Synthesis Mechanism	Input	Output
Propane/AT	Constraint-solving.	Propane Language	BGP configurations
SyNet	Constraint-solving.	Stratified Datalog	Router configurations
NetComplete	Sketch and CEGIS	Partial configurations (sketches)	Router Configurations
NetGen	Abstraction and constraint-solving.	Network Specifications (based on regular expressions)	SDN controller program
NetEgg	Programming By Example	Scenarios described w/ NetEgg Language	SDN controller program
Genesis	Constraint-solving and Divide-and-conquer to speed up synthesis.	"Tactics on labels" + Network Topology Considerations	Static router configurations
McClurg	Constraint Solving and CEGIS	*Petri Event Nets*	SDN controller program
Yanjun	Sketch; Programming By Example; Interactions with the Architect	Sketch + Scenario Preferences	Objective Function
Facon	Programming By Example	Input/Output Examples, NDLog Relation schemas	Declarative Networking Programs
Chipmunk	Sketch and CEGIS	(P4 or Domino) Sketch	P4 or Domino program
Lyra	Constraint-solving.	Lyra Language Program; topology configuration	Chip-specific code (NPL and P4)
Gallium	Rule-based compiler	Software NF (C/C++)	P4 program + x86 controller program
SyNAPSE	Modular Search	Software NF (C/C++)	P4 program + x86 controller program

Table 2.1: Comparative Table for all Related Tools.

3 Architecture

Contents

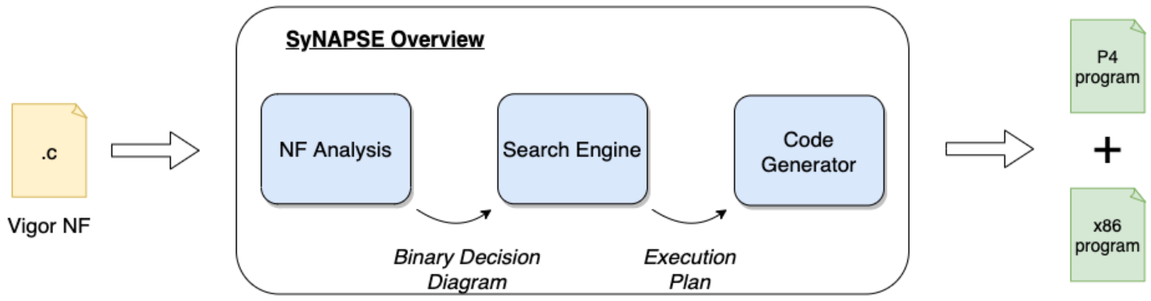


Figure 3.1: SyNAPSE architecture.

This thesis adds to the state of the art by presenting SyNAPSE, a tool capable of offloading packet-processing to a programmable software switch. It generates configurations for a P4 programmable platform and a x86 Controller from a network function (NF) programmed in the C language, in the context of the Vigor framework [6]. We use Vigor's static analysis tools to obtain a complete and sound representation of the NF. Then, by applying synthesis-based techniques to this output, we generate a semantically equivalent NF in the P4 language, which is compiled to a software switch with the goal to offload packet-processing functionality.

This chapter describes how SyNAPSE generates configurations for a P4 programmable software switch and a x86 controller. It is divided into 4 sections. First, we describe the architectural structure of our solution. Then, the subsequent sections are dedicated to each of the sub-components of SyNAPSE.

3.1 Overview

The SyNAPSE pipeline is composed of 3 main elements: Network Function Analysis, Search Engine, and Code Generator (Figure 3.1).

The Network Function Analysis component builds on two tools – Vigor and Maestro – to perform static analysis of the software NFs. Vigor (Section 2.3) is a software NF verification tool that explores the use of symbolic analysis to check NF behavior against a specification for correctness. These static analysis methods are used by Maestro [16], the second tool, to build a sound and complete representation of the NF behaviour. Maestro uses the NF representation to automatically parallelize sequential implementations of NFs, distributing traffic across cores and preserving the semantics of the sequential implementation. We, on the other hand, adopt this complete representation of the NFs behaviour to apply synthesis-based techniques in the Search Engine component. This component outputs an intermediate representation of the NF from which we are able to generate code for both the P4 and C programs. The resulting programs, produced by the Code Generator component, are semantically equivalent to the original NF and offload as much packet-processing functionality to the P4 switch as possible. SyNAPSE generates the P4 code by mapping function calls of the libVig data structures from Vigor directly to P4 tables. The SyNAPSE prototype successfully generates implementations for 5 Vigor NFs: a NOP, a static-bridge, a Learning Bridge, a Firewall and a NAT.

In the following sections we analyse how the remaining elements that take part in the NF functionality are mapped to P4 code.

3.2 SyNAPSE in Detail

SyNAPSE is able to generate configurations from software NFs written in the context of the Vigor framework. To understand how each SyNAPSE component works, it is necessary that we are familiarized with the Vigor data structures and other Vigor implementation details. In this section we also introduce a practical example to provide a better insight of how each of the SyNAPSE's components work in conjunction towards the goal of offloading packet-processing functionality to a P4 programmable software switch.

3.2.1 Vigor Data Structures

Vigor is a software stack and toolchain for building and running NFs that are guaranteed to be correct while preserving competitive performance and developer productivity (Section 2.3). To implement stateful parts of the NF code, NF developers can utilize a set of validated data structures. The three most relevant data structures used to implement network functions with Vigor are the `map`, `vector` and `dchain`. For each of the Vigor data structures we briefly explain the most common auxiliary functions, used by the programmers to manipulate these data structures.

`map`. The `map` data structure is a consistent hash-table which stores integers associated with a hashed key, of arbitrary type. To add an entry to a given map, the programmer can utilize the `map_put` method. To retrieve a map entry, one can use a key as an argument to the method `map_get`, which returns the corresponding stored integer.

`vector`. The `vector` data structure is used to store arbitrary type data associated with an integer index [16]. Similarly to the `map` data structure, the `vector` libVig provides methods to insert new entries (`vector_allocate`) and retrieve data (`vector_borrow` and `vector_return`).

`dchain`. The *double-chain* is a double linked-list of integers. It has a limited number of integers available and allows users to allocate a new index and verify if it is already allocated. Every index is associated with a timestamp and can be rejuvenated by the user, otherwise the index times-out and becomes unallocated.

Vigor NFs use the `dchain` data structure simultaneously with the `map` and `vector` data structures to expire entries associated to a given flow that is possibly no longer being used. Typically, the index allocated by the `dchain` is stored in the other two data structures and used to check whether a specific index was allocated. This value is obtained with the method `dchain_allocate_new_index`. Additionally, Vigor offers methods to extend the life-time of a given entry, expire an entry or free an index.

The above mentioned functions often operate directly over specific header fields. In order to handle variable length headers the Vigor API allows NFs to access the packets in chunks, as opposed to a single buffer containing the whole packet. This allows the Vigor API to calculate which packet field is being passed as an argument by checking the number of packet chunks requests made before the call. When the symbolic execution is applied to a Vigor NF, every call that uses a specific header field as an argument, this argument will later be described with the `packet_chunk` symbol.

We are able to generate P4 code to NFs written in C by mapping directly calls to libVig functions to P4 tables. For example, the behaviour of a `map_get` function or a `vector_borrow` can be represented in P4 by a hit/miss in a P4 table. During the next sections we will investigate how other elements, that are part of the NF functionality, are mapped to elements of a P4 program or of a x86 controller program.

3.2.2 Running Example

We will use a simplified version of a Learning Bridge, whose functionality is represented in the pseudo-code in the listing 3.1. In order to correctly forward packets, the Bridge NF must first dynamically associate MAC addresses and ports. In this example we use the a single `map` data structure, that uses MAC addresses as key to an integer value that corresponds to a port.

When a packet arrives to the switch, two possible outcomes are possible: the packet is broadcasted (sent to all ports) or forwarded to a specific port. The latter only happens if both MAC addresses are known and mapped to a specific port connected to the switch. More concretely, the behaviour of the Bridge NF is the following:

1. The Ethernet header is processed and the source and destination MAC addresses are retrieved (line 4 of the listing 3.1).
2. If the source MAC address is not yet stored, a new table entry is added (lines 5 and 6 of the listing 3.1).
3. If the destination MAC address is NOT present, the packet is broadcasted (lines 9 and 10 of listing 3.1).
4. If the destination MAC address is present, then both MAC address are known and packet is forwarded `map` (lines 12 and 13 of listing 3.1).

During this chapter we study the modifications that each SyNAPSE component performs to the initial input in a **simplified manner**. The goal is to provide concrete examples that assist the theoretical explanation of each component.

Listing 3.1: Pseudo-code of vigbridge.

```
struct Map* map;

void process_packet(int device, pkt_t packet) {
    header_t p = get_ethernet_header(packet);
    if (!map_contains(map, p.mac_src)) {
        map_put(map, p.mac_src, device);
    }

    if (!map_contains(map, p.mac_dst)) {
        broadcast(p);
    } else {
        int dst_device = map_get(map, p.mac_dst);
        forward(p, dst_device);
    }
}
```

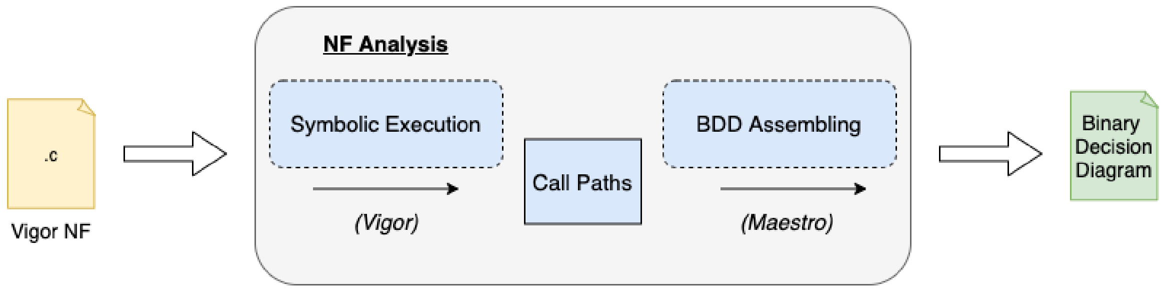


Figure 3.2: NF Analysis Module architecture.

3.3 Network Function Analysis

In this module, the main goal is to generate a Binary Decision Diagram (BDD) that represents all abstract functionality of the NF. The components of this module can be visualized in Figure 3.2. We analyse the NFs code using an exhaustive symbolic execution engine to generate a set of Call Paths. These Call Paths represent all the possible paths through the code triggered by a packet's arrival [16]. The Call Paths are then used to progressively assemble the binary decision diagram (BDD) containing all the NF functionality.

3.3.1 Call Paths

As mentioned in Section 2.3 Vigor written NFs are built on top of DPDK, an open-source framework built for fast packet processing that offers a set of software libraries and drivers. They are formally verified by Vigor, meaning they are guaranteed to be semantically correct, memory-safe and crash-free. The goal of this sub-component is to generate a complete and sound representation of the NFs behaviour, using KLEE symbolic execution engine.

Listing 3.2: Call paths generated for the example NF by the symbolic execution engine.

```
1  1: !map_contains_0 && !map_contains_1
2      => get_ethernet_header(packet)
3      => map_contains(map, p.src_port) -> map_contains_0
4      => map_put(map, p.src_port, device)
5      => map_contains(map, p.dst_port) -> map_contains_1
6      => broadcast(p)
7
8  2: map_contains_0 && !map_contains_1
9      => get_ethernet_header(packet)
10     => map_contains(tb, p.src_port) -> map_contains_0
11     => map_contains(tb, p.dst_port) -> map_contains_1
12     => broadcast(p)
13
14 3: !map_contains_0 && map_contains_1
15     => get_ethernet_header(packet)
16     => map_contains(map, p.src_port) -> map_contains_0
17     => map_put(map, p.src_port, device)
18     => map_contains(tb, p.dst_port) -> map_contains_1
19     => map_get(tb, p.dst_port) -> map_out_0
20     => forward(p, map_out_0)
21
22 4: map_contains_0 && map_contains_1
23     => get_ethernet_header(packet)
24     => map_contains(tb, p.src_port) -> map_contains_0
25     => map_contains(tb, p.dst_port) -> map_contains_1
26     => map_get(tb, p.dst_port) -> map_out_0
27     => forward(p, map_out_0)
```

Each call path represents a possible outcome when processing a newly arrived packet. Thus, the set of all Call Paths obtained is able to depict the NFs full behaviour. A Call Path is composed of *calls* and *constraints*. The *calls* correspond to either calls to packet management functions, for example the `get_ethernet_header` in listing 3.1), or calls to libVig functions such as `map_get` and `map_put`. The constraints identify the call path by stating the conditions that must be true, or false, in order for a packet to trigger a particular code path and the corresponding calls. When a packet arrives, whenever all the criteria are met for a given call path, then what follows is the execution for every call specified in the call path.

When the symbolic execution is applied to the function `process_packet` of our running example (listing 3.1) we obtain the call paths observed in listing 3.2. Every time an *if condition* appears in the code, the symbolic execution engine will identify the constraint that discerns the two groups of call paths. In the lines 5 and 9 of the listing 3.1 we can observe the two *if conditions* in our NF, regarding the outcome of the `map` auxiliary function `map_contains`. The function is used to tells us if there is a pairing between a port and the respective MAC address.

Each *if condition* has two possible outcomes, so the symbolic execution engine will generate four possible call paths, represented in the lines 1, 8, 14 and 22 of the listing 3.2. In this listing 3.2 each call path is identified by an integer followed by their symbolic constraints and a list of function calls. The symbolic constraints correspond whether a pairing between the ports and the source/destination MAC addresses exists.

Keep in mind that in a less simplistic case, the complexity of a Learning Bridge NF would have generated more detailed call paths, with more data structures involved, and the complexity of the nodes would be superior, as they would reflect all the manipulated symbolic values.

3.3.2 Binary Decision Diagrams

After generating the call paths for the NF program in listing 3.1, we resort to Maestro's work to analyse them and assemble the Binary Decision Diagram [16]. This component outputs a BDD that represents all possible outcomes for an incoming packet.

The nodes in the BDD represent calls to packet management functions, calls to libVig functions and conditional statements. While the conditional statements in the BDD simply contain a KLEE expression of the discriminating constraint, the remaining BDD nodes contain data such as the function name and its call arguments, as well as selected byproducts of the symbolic execution. The stored KLEE expressions contain symbols that vary depending on the Vigor API calls and their corresponding arguments. It is important to retain that every node describes a specific functionality represented in the NF code. For example, a BDD node representing the *map_get* Vigor call would indicate the name of the call, the *map* ID and the KLEE expression that resulted from the symbolic execution.

The process of assembling the BDD starts with the iteration over all the generated call paths at the same rate. After each iteration the algorithm will compare the calls between all processed call paths. If all the call paths share the same call, a new corresponding node is added to the tree. If this requirement the algorithm proceeds to identify the discriminating constraint that discerns the two groups of call paths and adds a conditional node to the BDD. Whenever a conditional node is added to the tree, it bifurcates into two subtrees: one corresponding to the program flow in which the condition is true, and the other to the condition in which the condition is false. Then, to each one of the trees, we make recursive calls to the algorithm. The process is repeated until, eventually, all call paths are processed.

Now that we understand the algorithm that assembles BDDs, let us consider the call paths that were generated by the symbolic engines for our example Bridge NF. The call paths represent all the possible outcomes for a packet when processed for our NF, so when we apply the BDD assembling algorithm, the BDD will represent the complete functionality of the original NF code. As the algorithm iterates over all the call paths at the same rate, it first checks if the first call of all the call paths is the same. In listing 3.2 we verify that all call paths share the same call. As such, the algorithm adds the corresponding node for the call `map_contains( tb, p.src_port )` checking whether the source port of a packet is already paired to a MAC address in the map *tb*.

In the second iteration the BDD assembling algorithm encounters two different calls in the set of call paths. While the call paths 1 and 3 are followed by the call `map_put`, the remaining are followed by the call `map_contains`. For the two groups of call paths the algorithm would find the discriminating

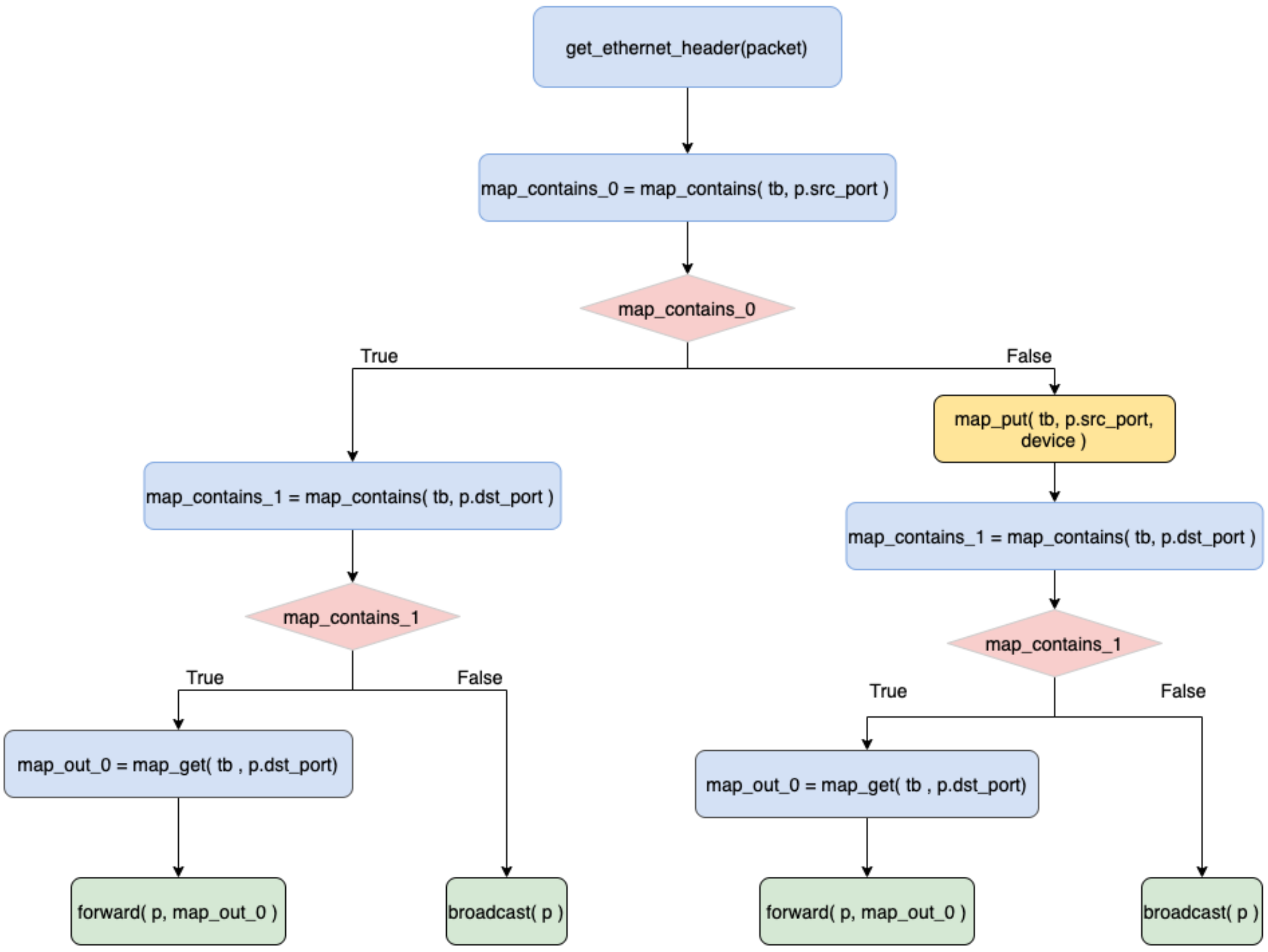

get_ethernet_header(packet)
map_contains_0 = map_contains(tb, p.src_port)
map_contains_0
True
False
map_put(tb, p.src_port, device)
map_contains_1 = map_contains(tb, p.dst_port)
map_contains_1
True
False
map_out_0 = map_get(tb , p.dst_port)
forward(p, map_out_0)
broadcast(p)
map_contains_1 = map_contains(tb, p.dst_port)
map_contains_1
True
False
map_out_0 = map_get(tb , p.dst_port)
forward(p, map_out_0)
broadcast(p)

Figure 3.3: Generated binary decision tree for the call paths in listing 3.2.

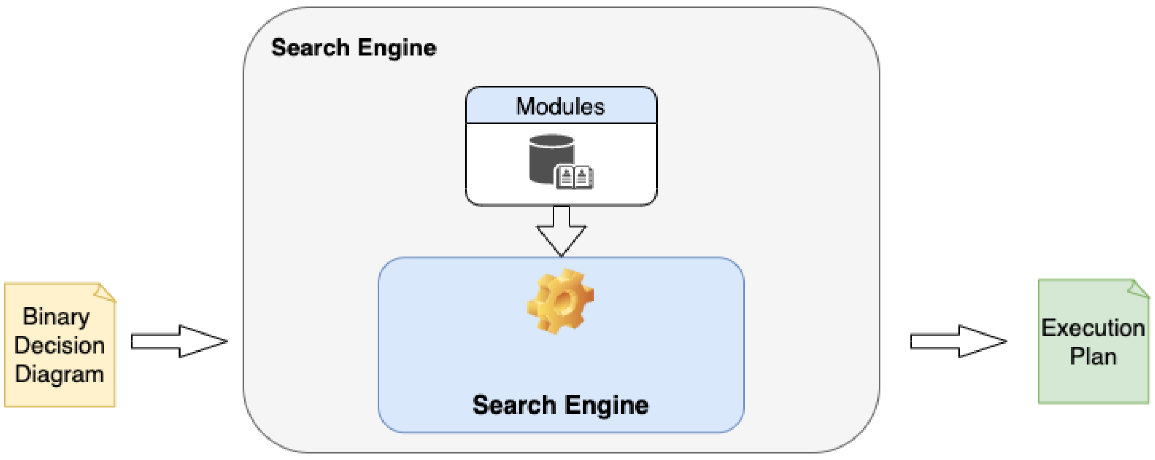


Figure 3.4: Search Engine Module architecture.

constraint and add a conditional node to the tree, separating the tree in two scenarios: one where the constraint is verified and other where the constraint is false. The algorithm would then be applied to each sub-tree, adding new nodes if calls are the same between all nodes of a sub-tree or recursively calling the algorithm whenever conditional nodes need to be added.

If we analyse the call paths we can observe that the discriminating constraint corresponds to the result of the `map_contains` function call. In the context of the example Bridge NF, this result corresponds to the presence of the Ethernet source address in the map *tb*. At this point the algorithm would be applied recursively to the two sides of the branch and the process is repeated until the algorithm iterates through all call paths. When the sub-trees are fully processed they are attached as branches to the original BDD. The resulting BDD for the example NF is described in Figure 3.3. The four green nodes correspond to the possible outcomes, this was expected as the symbolic execution engine generates four call paths.

3.4 Search Engine

In this section we analyse the search methods and the algorithms employed by the SyNAPSE Search Engine, in order to analyse the BDD and produce an Execution Plan (EP) - a tree structured intermediary representation of the NF functionality.

While the BDD only represents the abstract functionality and calls required to implement the NF logic, the intermediate representation achieved in this component can describe concretely the set of operations necessary to the x86 controller and P4 target platform.

We created the Module entities to describe all of the available operations for both our targets: the x86 controller or the P4 Software Switch. The Modules allow the developers to add support for other functionality, or adding support for new targets, by creating new Modules. During the search process, we progressively replace the nodes in the BDD if any of the existing Modules matches their described functionality. The BDD node is replaced by an Execution Plan Node incorporating data elements of both the Module and the BDD node.

Considering our target, the x86 controller and the P4 Software Switch, the output of this component consists in a combination of EP Nodes associated to the P4 Software Switch and the x86 Controller. From the resulting multiple combinations of EP Nodes we choose the EP that is best aligned with our goal of offloading the packet-processing functionality to a P4 target. The strategy we use to estimate which combination of nodes is the best solution, consists in a heuristic mechanism that compares EPs and outputs the EP that has the greater number of EP Nodes that target the P4 Software Switch. By prioritizing solutions with the highest count of P4 Switch nodes in an Execution Plan, it is expected that the final NF implementation will be more reliant on the P4 target than the x86 controller. Given that most of the NF functionality can be implemented in a P4 Switch, we can expect that the initial nodes of the EP will target the P4 switch. Eventually, after a few algorithm iterations, the algorithm will find a node that can only be processed by a x86 controller. When there is a change in the targets of the EP nodes, from P4 Modules to x86 Modules, additional attention is required. We will address this challenge in more detail later in this section. We have determined that for more simple NF implementations the exclusive use of P4 switch modules to represent the functionality in a BDD is possible. However, stateful NFs always require an interaction of both the P4 switch and the controller, thus the EP solution is always a mix of Modules.

Our Search Engine receives as inputs: a BDD and a pool of Modules consisting in all the available modules for the P4 Software Switch and the x86 controller targets, and outputs an Execution Plan where the nodes specify the operations required from the P4 Software Switch target and the x86 controller target in order, creating a functionally equivalent implementation to the one represented in the BDD.

3.4.1 Search Space Algorithm

Let's analyze how the Search Engine algorithm finds a combination of Modules that can implement all the operations specified in the BDD nodes. As mentioned, the Search Engine receives as inputs: the BDD and a pool of Modules (which consists in all the available modules for the considered targets). The initial Execution Plan will have the same structure as the BDD. For each BDD node, the algorithm will find a Module that can represent the behaviour described. When a Execution Plan replaces the BDD node by a Module, the Module is added as an Execution Plan Node to Execution Plan. During the execution of the algorithm, multiple EPs will be generated from the initial EP in order to explore different solutions, so we always keep track of the next BDD node to be processed for each of the EPs. The algorithm can be explained in three phases: sorting Execution Plans, Module processing and EP construction. During the search process multiple Execution Plans can be created.

In the *sorting phase* a set of partial EPs is sorted according to the Heuristic, that compares multiple EPs according to the overall number of nodes and of P4 switch Modules. This phase will return the preferred EP that will be analysed by the Module Processing phase.

The *module processing phase* is responsible for finding Modules that can implement the functionality described in the BDD nodes. In this phase every existing Module will try to process the current BDD node, considering the context in which the BDD node is being processed, i.e. it analyses all the Modules that have been previously added to the Execution Plan. Thus, different contexts can cause that a BDD node is processed in a distinct manner.

In the third phase of the algorithm, *EP construction*, for every Module that is able to process a BDD node, we create a new Execution Plan and replace the BDD node with the corresponding Module. When the Modules replace the BDD nodes, they are added as an Execution Plan Node, inheriting all the expressions generated by the KLEE symbolic engine. Later these expressions will be essential to generate the code for the P4 and x86 programs.

At this point we apply a node reordering algorithm to the newly created Execution Plans. In this algorithm we identify possible rearrangements and create new EPs, which are then finally added to the Search Space. Although these node rearrangements do not always lead to better solutions, they alternative scenarios to be explored. The reordering algorithm can perform two kinds of rearrangements:

1. Reorder consecutive nodes that have no dependency with each other.
2. Identify a common node (corresponding to a call or a conditional statement) that is used in all code paths in a given sub-tree. This node is removed and placed before the first common node of the sub-tree.

Once all modules try to process a BDD node, and we have created all the subsequent EPs, the EP that was initially picked in the *sorting phase* can be removed from the search space as all possible solutions have been explored. The algorithm moves to the next iteration.

3.4.2 Implementation Details

In the Search Engine one operation described by a BDD node can be implemented in various ways and by multiple targets. This contributes to the number of solutions that constitute the Search Space, but what creates the explosion in the number of solutions are the EPs created by the node reordering algorithm. This becomes more evident as the complexity of the Vigor NFs increases. At the time of writing, the prototype's Search Engine is not capable of pruning the large search spaces efficiently as the implemented heuristics are to naive, which translates into unfeasible times to completely explore all solutions generated.

Optimization. We have developed a mechanism that allow us to reduce the number of generated

P4 tables (that are a byproduct of a *map_get* call) by merging tables that originally accessed the same *map* data structure. With this optimization, even though the exact same number of table entries need be introduced, we are able to reduce complexity on the controller side as we don't need to determine in which table an entry should be added. We try to perform this optimization during the execution of the search algorithm whenever a *map_get* call is found. We use the *Table_Lookup* Module to replace the BDD node call and merge it, if the same *map* was already stored in the past.

Module Types. As mentioned previously, the BDD nodes contain information about the condition or call they represent, such as the libVig data structure ID, the libVig function name, the arguments associated and the symbols they manipulate. In order for the Modules to replace BDD nodes in the Execution Plans, the algorithm will try to match all the Modules against the BDD nodes. For these matches to happen, the Module presents a number of assertions that must be cleared, usually including the supported libVig functions, validating the header field values and verifying the libVig call arguments.

Before studying how the Search Engine algorithm acts, we first need to investigate the Modules that will replace the BDD nodes and that take part in the Execution Plans. We will focus on the ones necessary to explain the running example, but also other common and important Modules. The Module **Table_Lookup** can replace `map_get` and `vector_return` calls. These calls to libVig data structures can be represented as a P4 table as they simply refer to a table inspection. In this module we also implement the table merging optimization mentioned above. However, the Module **Table_Lookup** can not replace calls like `map_put` and `vector_borrow`, as from a P4 standpoint, they represent adding new table entries. Since this behaviour can only be implemented by a controller, we replace these calls with a **Send_to_controller** Module that is responsible for shifting the NF implementation target, that up to that point was being implemented by P4 switch Modules. As mentioned before, Vigor accesses packets in chunks in order to support packets with different header length sizes. Vigor allows user to interact with packet headers using sequential calls to the `packet_borrow_next_chunk` call. The Search Engine will replace these calls to packet management functions with specific Modules depending on the number of existing calls to the "chunks". For example, when the the first `packet_borrow_next_chunk` call is identified, the algorithm will replace it by a Module that will retrieve all necessary header fields of the Ethernet header, the second call would be replaced by a Module targeting the IP protocol and the third call would be substituted by a Module targeting the layer 4 protocols, TCP and UDP. Lastly, the BDD nodes that represent conditional statements are simply replaced by Conditional Modules that create two diverging branches and keep the same KLEE expression.

3.4.3 Running Example

Now that we understand all elements of the search algorithm and its general methods of operation, let's apply all these concepts to our running example. For sake of understanding, we will only explain the first iterations of the algorithm, complementing the explanation of the Search Engine component with important insights and details.

Let's first analyse the inputs to our Search Engine: the BDD generated by the previous NF Analysis component (fig. 3.3) and the pool of Modules that target the P4 Software Switch and the x86 Controller.

In the BDD (fig. 3.3) we can observe different libVig function calls that operate over the same `map` data structure. We can easily deduct that a single `map` is going to appear in the P4 program as all the calls related to this data structure access the same data. We can also predict that resulting EP will include nodes that target the P4 Software Switch and the x86 controller, as there is one `map_put` node in the BDD, whose functionality can only be implemented by a controller, as it represents adding a new table entry in our P4 table.

The resulting EP is represented in fig. 3.5. The projections are evident in the figure but let's address the fact that the number of nodes in the EP is substantially different from the BDD. Looking at fig. 3.5 we can verify that what follows node 5 (**Table_Lookup**) is not the result of processing the next BDD node, but the entire original BDD targeting the x86 controller. The reason is the need of our next SyNAPSE component (Code Generator) to split the Execution Plan according to the nodes target platforms. This way, the Code Generator can correctly generate the x86 controller code in conformance with the NF logic.

In the first iteration the algorithm starts with an Execution Plan that has no Modules. At this point, the sorting phase is trivial as no BDD node has been processed, so we move to the second phase, where each of the available Modules will try to process the first node in the BDD. The first node of the BDD corresponds to a packet management function responsible for verifying the Ethernet Header fields. Both our targets can process this node, resulting in the creation of two single-node new EPs. After exploring all possibilities to process a module, the EP that was elected in the sorting phase can be discarded. In the second algorithm iteration, we first elect the EP with more P4 nodes. In the second step we verify which Modules can process the second node of the BDD `map_contains`. After discovering that it can be processed by two modules: an x86 **MapGet** and a P4 **Table_Lookup**, we move to the third and final step where two new Execution Plans are created.

This process of picking an existing partial EP, individually find Modules that can process a BDD node and creating EPs and adding them nodes, repeating itself until all nodes in the original BDD are replaced. By the end of this process we achieve a semantically equivalent NF representation, where the functionality is split between a P4 Software Switch and a standard x86 controller.

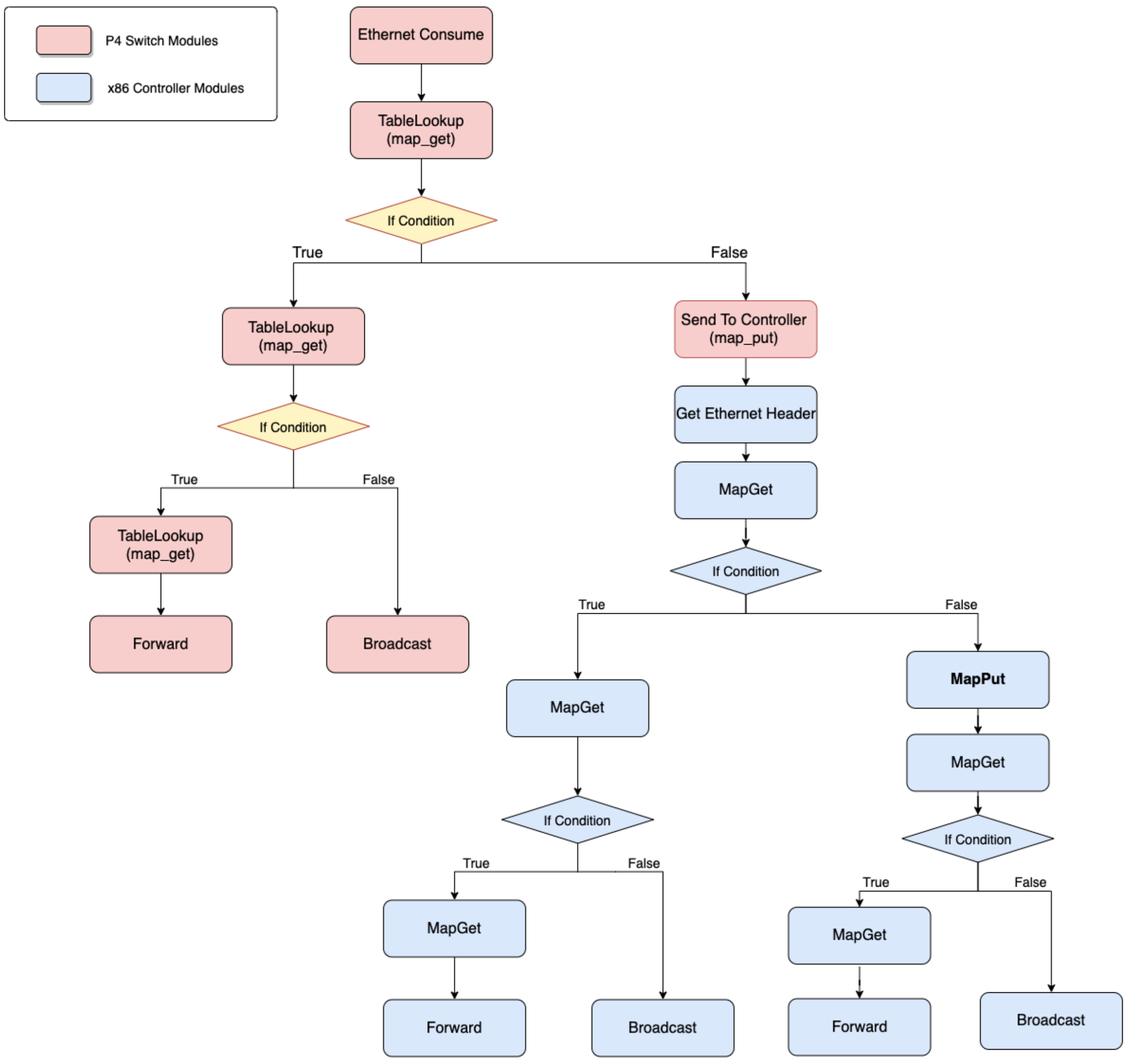


Figure 3.5: Resulting Execution Plan for the Running Example.

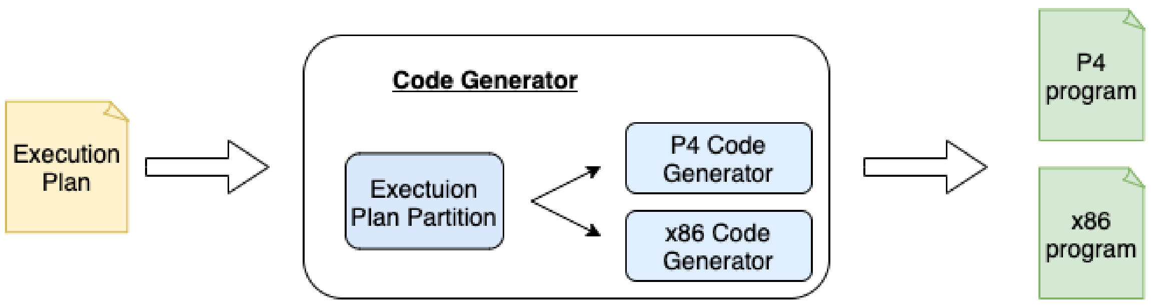


Figure 3.6: Code Generator Module architecture.

3.5 Code Generator

In the Code Generator component, shown in fig. 3.6, the goal is to process the Execution Plan and generate the P4 and x86 controller code. This is the only responsibility of the last SyNAPSE component, as all optimizations are performed by the previous modules.

The first task of this component is to partition the Execution Plan according to the target platforms of each node. Considering our two targets (the x86 controller and the P4 switch), two Execution Plans are generated. The information contained in these EPs is gathered and used to generate code for the two programs. In each program we start with pieces of code that are common to all NF implementations. In the P4 template program we define the global configuration of the pipeline for the software switch that is our target (the *V1model* switch, from p4.org). The generated code will complete the template program by defining the required packet headers fields and metadata fields, completing the parser and deparser logic and defining the ingress logic of the controller. In the latter we instantiate the tables and actions required to represent the behaviour described in the partitioned EP.

We apply the same process for the x86 controller program. The x86 controller program is used to complete the original controller used in Vigor, replacing the Vigor loop cycle (responsible for processing packets redirected to the controller) with a newly developed SyNAPSE Runtime API, an extra layer that complements the P4Runtime API. These Runtime APIs allow the network controller to manipulate data plane elements defined by a P4 program, such as tables, actions and externs like counters, meters and registers. The generated x86 controller code will complete the template code by declaring the required libVig data structures, additional requirements for the SyNAPSE Runtime API and the two main functions of the Vigor programs: *nf_init* which instantiates the required libVig data structures and the *nf_process* function that specifies all the NF controller logic. P4Runtime uses a Protobuf-based API to automatically generate gRPC client/server code for multiple languages.

3.5.1 Implementation Details

One of the main challenges for the Code Generator component dwells in maintaining state between all tables in the data plane. As mentioned before, we map libVig data structures (`map` and `vector`) operations to P4 tables. When multiple tables are being used in a P4 program it is required that all packets are processed according to the most recent state of every table. To avoid situations where a packet reads inconsistent states between the tables, we developed a versioning control mechanism implemented with an extra table that uses *tags* to identify different states. These tags individually track the version of all tables by incrementing its value every time the controller performs a write operation. All SyNAPSE generated tables have a matching field with the corresponding version of the table. Since the *tag_control* table is the first table applied to a packet, the packet-processing action applied to the packet is altered if any of the tables does not match on the version indicated.

During runtime, the x86 controller program is essentially used to add or expire new table entries. The interaction between the control plane and the data plane is performed via Protobuf messages of the P4Runtime API mentioned before. To add new table entries we resort to P4Runtime's Packet I/O framework which allows to capture additional information by defining supplementary P4 headers. This framework allows to specify additional information such as the original data plane port where the packet was received, the timestamp when the packet was received or if the packet is a clone. These special P4 headers are required for the P4runtime server to process packet-in and packet-out stream messages.

We define a function *send_to_controller* that receives a *code_id* parameter and assembles the *packet_in* header. Depending on the logic of the NF, a packet can be sent to the controller in different situations during the execution of the P4 program. For example it might be required that the controller adds a new table entry to a distinct table, or to a specific group of tables. In order to handle these multiple scenarios and trigger the correct behaviour in the controller, each *send_to_controller* of the program passes a different *code_id*, then we generate the different scenarios to handle the packets in the controller accordingly.

To generate the controller and the P4 program, the Code Generator will first analyse the partitioned EPs nodes. The information gathered from the EP nodes is categorized and subsequently used to complete templates of a P4 program and a C controller program. These templates consist of lines of code that are common to all NF implementations. In the P4 program template we define keywords, the headers that take part in controller interaction, general state transitions and packet-processing actions that are responsible to send the packets between ports (e.g *forward* and *send_to_controller*). Additionally, we identify specific areas of the template program that will be completed with the information retrieved by the Code Generator.

Considering the code generation for the P4 program, the information retrieved in the EP nodes is categorized according to the forwarding pipeline stages that the nodes will be part of. The nodes can be used to gather data for P4 structures, such as metadata, tables and headers, or even pipeline stages like the parser and ingress stages. In some occasions, the information in the EP nodes can be used to complete more than one of these structures. For instance, the EP nodes that correspond to conditional statements are used in the parser stage and in the apply block of the ingress stage. In the parser stage we generate the stage transitions by organizing the conditions according to the the logic outlined by the EP node, to check whether the packet header fields can be used to generate the conditions designated in the EP node's KLEE expressions. If we are not able to generate these conditions the P4 program immediately rejects the packet or sends it to the controller, thus avoiding further processing of these packets. In the ingress stage the P4 code generated for the conditional statements will be used to navigate the full logic of the EP (e.g applying tables, forward and drop actions), instead of transitioning to other parser stages. We were able to achieve P4 code that followed syntax and semantic rules after some iterations of trial and error, during which we found limitations and features of the P4 compiler. Some of these findings required to adapt and refactor our code base to generate better P4 code.

In the Search Engine (section 3.4) we were able to produce an Execution Plan that depicted the behaviour of our example Bridge (fig. 3.5). To finally generate the code for the Bridge example implementation, first we need to partition the EP taking into account the backend of each node (x86 controller and P4 switch). At the end of this stage we have one EP that only targets the controller and one EP that targets the P4 software switch.

To generate the code for each backend program, the Code Generator traverses each of the EPs, retrieving and categorizing information of the EP nodes. Then, for the controller program, we analyse the number of tables to be created and the circumstances in which each table can be called. For the P4 program we collect information for the all pipelines, defining the P4 table that corresponds to the *map* data structure and the actions that are responsible for forwarding the packets when there is a MAC address and port pairing registered in the P4 table. The most relevant task consists in the code generation for the apply block of the ingress stage, where the NF logic is defined and the packet is processed.

3.6 Summary

In this section, we detailed the design and implementation of SYNAPSE. We described in detail how the three components of the current prototype manipulate their inputs in order to generate valid NF implementations. In the first module, we apply static analysis methods to Vigor software NFs to obtain a complete and sound representation of the NFs functionality, in the form of a Binary Decision Diagram. The nodes that compose the BDDs describe the libVig calls and conditional statements that need to occur in order to represent the behaviour of the original NF. This intermediary representation of the NF allows us to apply synthesis-based techniques to find NF implementations that offload the packet processing functionality to a P4 software switch. The Search Engine module includes a heuristic that is capable of choosing a solution based on the characteristics of the intermediary representation. Finally, in the code generator component we translate the representation of the NF to a C program and a P4 program, that correspond to a semantically equivalent implementation of the initial NF, targeting an x86 controller and a P4 programmable software switch. We discussed the map merging optimization implemented, the algorithms employed, an their impact in the search space of our particular problem.

In the next chapter we evaluate the NFs generated by the SyNAPSE prototype when subjected to real network traffic.

3.6.1 Contribution

The SyNAPSE prototype is being developed as part of a CMU-PT project, and as such is a joint effort. In this closing section I will detail my contribution in this thesis.

My first contribution to the SyNAPSE prototype started with a serialization and deserialization mechanism for the BDD. Before developing this mechanism, the NF Analysis module would need to generate the Call Paths and assemble the BDD before any Search Engine operation. As this process was repetitive and time-consuming, my mechanism enabled reducing the total time to generate solutions. However, because we made structural modifications to the call paths, the BDD nodes and their symbols, new issues had to be addressed. The main complications introduced by modifying the BDD symbols required changes in the algorithm of node reordering, auxiliary functions for the search engine and the map merging mechanisms.

In the Search Engine component I was involved in the creation of the modules for the target platforms, with focus on the P4 Switch Modules. I focused on code refactoring of the existing SyNAPSE structure to simplify the overall system complexity, discarding redundant structures. Additionally, for the `Table_Lookup` module I contributed to the implementation of the table merging mechanism explained earlier in this chapter 3.4, by developing auxiliary functions which allow to find `map` data structures that could be merged during the execution of the Search Engine.

In the Code Generator component I implemented the partitioning of the EPs according to the backend target, the target Code Generators, and the heuristic scoring system. My contribution included testing the generated code. During this phase, we discovered limitations of the P4 compiler and fixed some bugs, which we will share with the community.

Finally, I performed the evaluation of the SyNAPSE prototype, which will be described in the next chapter.

4 Evaluation

Contents

SyNAPSE is able to generate semantically equivalent NFs of five Vigor NFs: a NOP, a Static Bridge, a dynamic Learning Bridge, a Firewall and a NAT.

We have manually developed P4 NF implementations equivalent to the Vigor NFs. These hand-written implementations where designed to offload as much as possible packet processing to the P4 switch. With these implementations we intend to emulate non-expert P4 developers and compare them to SyNAPSE generated implementations. Since SyNAPSE generates implementations for a P4 software switch, we are unable to provide precise performance metrics. As an alternative, we evaluate the generate NFs by assessing how they interact with the controller, as this is a proxy for performance: the more packets are sent to the controller, the less performant is the resulting NF.

We aim to answer to two main questions:

1. How much time does it take for SyNAPSE to generate implementations for every NF?
2. How do hand-written NFs and SyNAPSE generated NFs compare in terms controller interaction (number of packets sent to the controller)?

The first section of this chapter describes the Vigor NFs from which the prototype can create implementations, followed by a discussion of the setup and environment in which the NFs were evaluated, and consequently a section in which we present and discuss the results obtained. In this section we also estimate the throughput of the implementations generated by SyNAPSE if they were to be deployed in a real production-leve hardware switch (such as the Intel Barefoot Tofino). The last section of this chapter compares the generated NFs with NFs developed by hand using P4 externs, to simulate an experienced P4 programmer.

4.1 NF Description

SyNAPSE is able to generate implementations for 5 Vigor NFs: a NOP, a Static Bridge, a Learning Bridge, a Firewall and a NAT. The static analysis methods we employ allow us to obtain complete and sound representations of the Vigor NF code. The applied synthesis-based search allows us to generate NFs that are semantically equivalent to the ones defined in Vigor.

In this section, we explain the behaviour of each Vigor NF from which we have generated an hand-written NF implementation. Additionally, we analyse the number of call paths generated and the time it takes SyNAPSE to synthesize NF implementations. All NFs were designed and adapted to fit the topology described in fig. 4.1. In the following descriptions we use the terminology LAN and WAN to refer to the ports each host is connected to.

NOP. This NF is a simple stateless NF that forwards packets from the LAN port to the WAN port and vice-versa. Takes SyNAPSE less than one second to generate a NOP implementation. The resulting code consists in single P4 program, as no controller interaction is required for this NF.

Learning-Bridge associates MAC addresses to ports. Initially the Bridge does not know any pairing and, as new connections reach the NF, it dynamically learns new associations forwarding the following packets according to its the registered associations. In the hand-written version we use a table to register port-MAC addresses connections. The prototype successfully generates an implementation in 159 minutes and 21 seconds and the total search space consists in 1453 solutions.

Static-Bridge. This implementation constitutes a static version of the Learning Bridge. In this version the MAC table that contains the MAC addresses to port associations is filled at configuration time. SyNAPSE finds 29 possible solutions and takes 1.6 seconds to find and generate an implementation for the Static Bridge. Similarly to the NOP NF, there are no packets sent to the controller, thus the resulting code is entirely ran on the P4 Software Switch.

Firewall. The Vigor's firewall implementation only allows bidirectional communication if the connection is initiated from the LAN side, using a map to store flows. Flows originated in the WAN and that were not previously registered are dropped. To check whether a packet received in the WAN side is already in the map, the TCP/UDP ports and IP addresses are inverted. We use a similar strategy in the hand-written NFs that need to operate with 5-tuple flows (Firewall and NAT). For each connection we register two different table entries, switching the layer 4 ports and the IP addresses. For the SyNAPSE prototype it takes 541 minutes to iterate over the entire search space, which is composed of more than 33000 solutions.

NAT. A Network Address Translator (NAT) translates addresses between network addresses spaces. This NF allows communication between devices inside a private network with devices on the Internet through a single public IPv4 address. Vignat, a NAT implementation in Vigor, stores flows coming from the LAN and allocates a port that is used to index that flow. When a reply packet is sent to that port, vignat checks if the IP source address and TCP source port matches the IP destination address and TCP destination port of the stored flow, translating and redirecting it to the LAN if this check holds true [16]. SyNAPSE takes close to 298 minutes and 45 seconds to iterate over all 17875 solutions and generate a NAT implementation.

4.2 Benchmarking methodology

In this section I explain the general setup of the environment for the tests. All the results were obtained by running SyNAPSE in a virtualBox Virtual Machine with 10 cores and 16GB of RAM.

We use mininet to emulate a network topology composed of two hosts connected to a P4 software switch, as seen in fig. 4.1. For the P4 software switch we opted to use the *Simple Switch GRPC* implementation of the *Behavioural Model version 2* (BMv2), the reference software switch used to testing and developing P4 implementations. The advantages of this implementation is that its architecture allows to establish TCP connections with an external controller using the P4Runtime API. We use the P4Runtime API in order to manipulate tables for the hand-written NFs and a newly developed SyNAPSE Runtime API, an extra layer that complements the P4Runtime API, to manipulate the table instances defined in the SyNAPSE generated implementations. In the context of the tests we have performed, the main use for these APIs consisted in inserting and expiring table entries in the P4 tables.

In order to evaluate all the NF implementations we subject them to different traffic distributions and measure the number of interactions with the network controller. This indicator will later be used as a performance estimator, as gathering more accurate indicators such as throughput and delay is not accurate with a P4 software switch. For all the experiments, we used the *tcpreplay* tool to inject traffic in the network, from the Host 1 (LAN) to the Host 2 (WAN) and traversing the P4 Switch. The *tcpreplay* tool allows to replay previously captured traffic. We tested the NF implementation against packet captures of 1 million packets with Uniform and Zipfian distributions, obtained by applying Zipfian and Uniform parameters to traffic samples collected in a university network [34, 35]. The packet captures that follow Zipfian distributions are more suitable to represent real-world Internet traffic, where a large percentage of the total traffic corresponds to a small percentage of flows.

During each experiment, we documented the number of interactions with the controller that were triggered by table entry requests. To add table entries, the P4 switch, uses the packet I/O stream messages mentioned in section 3.5. We define special headers annotated with *controller_header* notation, that contain relevant data for the controller to add new table entries. In each experiment we vary the time that it takes the controller to expire table entries that timeout. To expire table entries the controller compares the timestamp of the last match for each and every table entry with the timeout defined for a given table. When the controller verifies that the timeout has expired, it sends an *IdleTimeoutMessage* containing the table entry *IDs* that have expired. This notification message is forwarded to the P4 switch which is responsible for clearing the table entries.

In the packet capture with Zipfian distribution there are a total of 8386 flows and 77 of those flows corresponds to 80% of the total traffic, whereas the packet capture with the uniform distribution there are a total of 34000 different flows and 25000 of them correspond to 80% of the total traffic. The hand-written implementations we designed for these experiments do not use any P4 externs to optimize the program. In a general case, these implementations only access the controller when a new flow is required to be added to a table.

4.3 Performance Benchmarking

In this section we evaluate the obtained results and compare the hand-written NFs against the NFs generated with SyNAPSE, in terms of percentage of packets that required interaction with the network controller. We collected statistics for the five available NFs and conducted four distinct experiments for

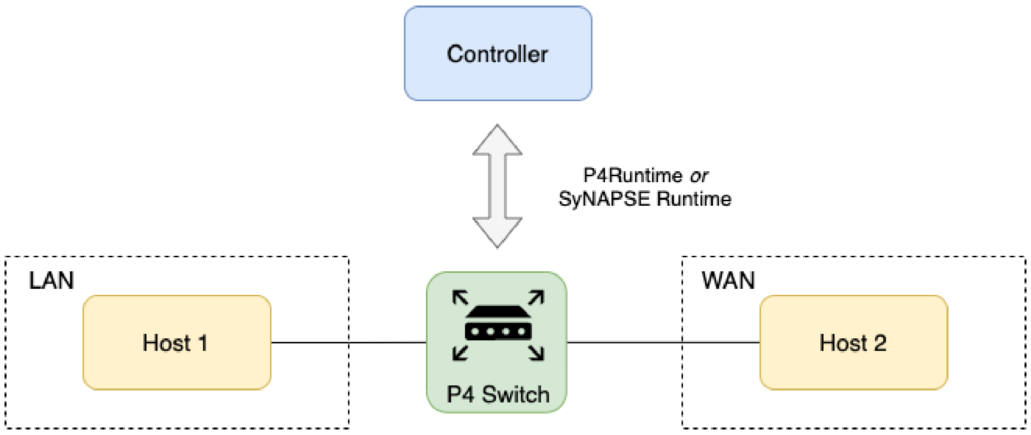


Figure 4.1: Simplified view of the network topology.

each NF, with timeouts of 15 and 45 seconds in each experiment.

The graphs in the figures 4.2, 4.3, 4.4, 4.4 compare the controller interaction for all SyNAPSE generated NFs and Hand written NFs subjected to both traffic distributions and considering expiration times of 15 and 45 seconds. The NOP and Static-Bridge NF do not require any controller interaction as these NFs do not require maintaining any state regarding flows. The NOP simply forwards packets from one switch port to another and the Static-Bridge previously parses all the flows that traverse the NF, adding them to a static table where no table entries are expired. For the remaining stateful implementations, in each experiment, the results obtained from both SyNAPSE generated and hand-written implementations are similar. As expected, we see that the increase in the expiration times from 15 to 45 seconds translates into more packets being sent to the controller, with a 0.46% and 0.56% bump for Zipfian and Uniform distributions, respectively. This indicates that, like our hand-written implementations, SyNAPSE can generate implementations that only access the controller when processing a packet of a flow that has not yet been stored in the switch.

We also conducted experiments with the stateful NFs (Bridge, NAT, and Firewall) and additional expiration times of 30 and 60 seconds to further investigate the behavior of the NF implementations generated by SyNAPSE, as well as the consistency and reliability of the SyNAPSE Runtime API. The results of these experiments can be observed in the graphs of the figures 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11. We can see from these graphs that all of the experiments resulted in similar results for both hand-written and generated NFs. Every experiment repetition for any of the expiration times yields statistically similar results, giving us positive indicators of the accuracy and reliability of the SyNAPSE Runtime API. The impact of increasing the expiration times is more noticeable in the Zipfian Distribution packet capture (figures 4.7, 4.9, and 4.11), where the percentage of interaction with the controller diminishes when the expiration times increase. For the uniform distribution packet capture, this percentage stabilizes after the 30 second expiration time. The traffic with a Zipfian distribution has a higher packet count for a small number of flows which translates into smaller time intervals between packets of the same flow. When we increase the expiration time, the table entries timestamps are renewed more frequently, resulting in a reduction in the number of expired flows.

We developed SyNAPSE to generate implementations that follow the same NF packet-processing approach as our hand-written NF implementations, which were created with the goal of minimizing controller interaction as much as possible without using any type of P4 externs. We can infer that this goal was achieved given the statistically similar results of all of our experiments.

Throughput Estimation. The P4 programs that are compiled correctly are able to run at line-rate in physical equipment. In the following arithmetic operations, we estimate the throughput of the generated implementations if they were to be deployed in a Wedge Barefoot Tofino with 32 ports each with a 100 Gbps rate. We use a weighted average (eq. (4.1)) to estimate the throughput of the SyNAPSE generated NFs, using a *Controller Throughput* of 1 Gbps, as documented in [36] and a (*Controller Packet %* that fluctuates according to the collected results. Because we are estimating performance metrics for different NF implementations we need to adapt the *P4 Switch Throughput* accordingly. If we see that the NF implementations could be adapted so that all of the 32 ports of the Barefoot Tofino would be used, we consider the throughput value of 3.2 Tbps. This remains true for the generated implementations of the NOP and Static and Learning Bridge NFs. For the remaining NAT and Firewall implementations, we consider a maximum throughput of 100 Gbps, using only two ports (the LAN port and WAN port), just like the original Vigor NF implementations.

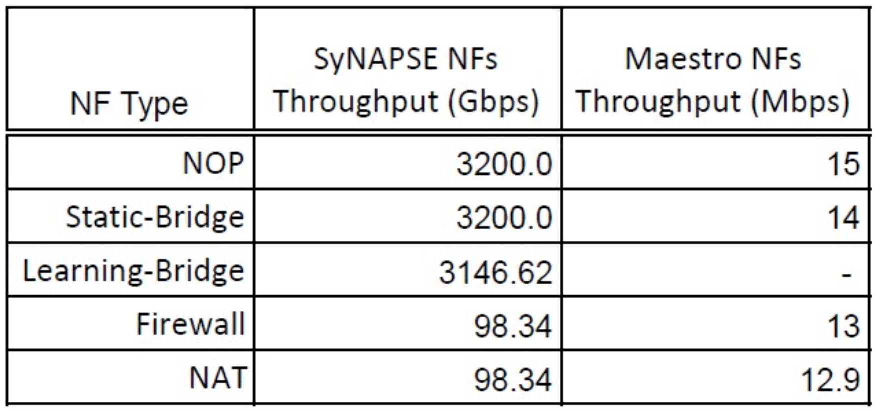
$$P4\ Switch\ Throughput\ *\ (1 - Controller\ packet\ \%) +\ Controller\ Throughput\ *\ Controller\ packet\ \% \tag{4.1}$$

To estimate the throughput of the generated NFs we use the results obtained in the experiment documented in fig. 4.4, with the table entry expiration time of 15 seconds and 1 million packets with a Zipfian Distribution, as this distribution can better describe real-world traffic scenarios. The estimate throughput for the stateless NFs like the NOP and the Static-Bridge, where no packet is sent to the controller, is 3.2 Tbps (eq. (4.2)).

$$3200 * (1 - 0.00) + 1 * 0.00 = 3200\ Gbps \tag{4.2}$$

In the Learning Bridge, we observe that 1,68% of the packets are sent to the controller, resulting in an average throughput of 3146,256 Gbps (eq. (4.3)).

$$3200 * (1 - 0.0168) + 1 * 0.0168 = 3146,257\ Gbps \tag{4.3}$$



NF Type	SyNAPSE NFs Throughput (Gbps)	Maestro NFs Throughput (Mbps)
NOP	3200.0	15
Static-Bridge	3200.0	14
Learning-Bridge	3146.62	-
Firewall	98.34	13
NAT	98.34	12.9

Table 4.1: Comparing throughput estimation for SyNAPSE NFs and the Maestro NFs.

The NAT and the Firewall implementations also reported that 1,68% of the packets are sent to the controller, thus the resulting throughput is 98,337 Gbps (eq. (4.4)).

$$100 * (1 - 0.0168) + 1 * 0.0168 = 98,337\ Gbps \tag{4.4}$$

These estimations indicate us that the NOP and Static Bridge can run at line-rate, as no packets are sent to the controller. The Learning Bridge estimation also has a positive outlook as it is close to line-rate processing values. The remaining stateful NF implementations were generated considering only the WAN and LAN ports, nonetheless we obtained estimated values close to the ceiling 100 Gbps.

We compared the throughput values obtained in the evaluation of the Maestro NFs [16], a multi-core optimized versions of the Vigor NFs [6], as demonstrated in the table 4.1. We found that the estimated values for the SyNAPSE NFs surpass the throughput values obtained in of the Maestro NFs shown in [16].

4.4 Discussion

In the previous sections we studied how SyNAPSE generates stateful NF implementations for a software switch. Initially, the scope of this thesis also included generating NF implementations that resorted to P4 externs to optimize the performance of the NFs, but due to time constraints we were unable developed the required modules to attain this goal. However, we manually developed hand NF versions that would be compared, in the evaluation section, to the SyNAPSE optimized implementations.

In these manually developed advanced versions of the NFs we focused on reducing the controller interaction by adapting P4 registers to store and match information, replacing the functionality of the traditional P4 match-action tables. The P4 registers are stateful memories whose values can be read

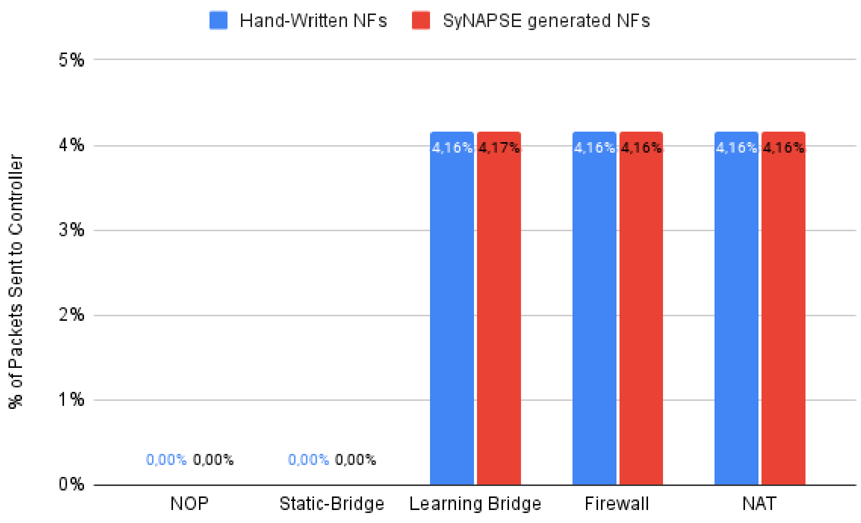


Figure 4.2: Comparison of the percentage of packets sent to the controller for all NFs, with a 15 second expiration time and 1 million packet with Uniform distribution.

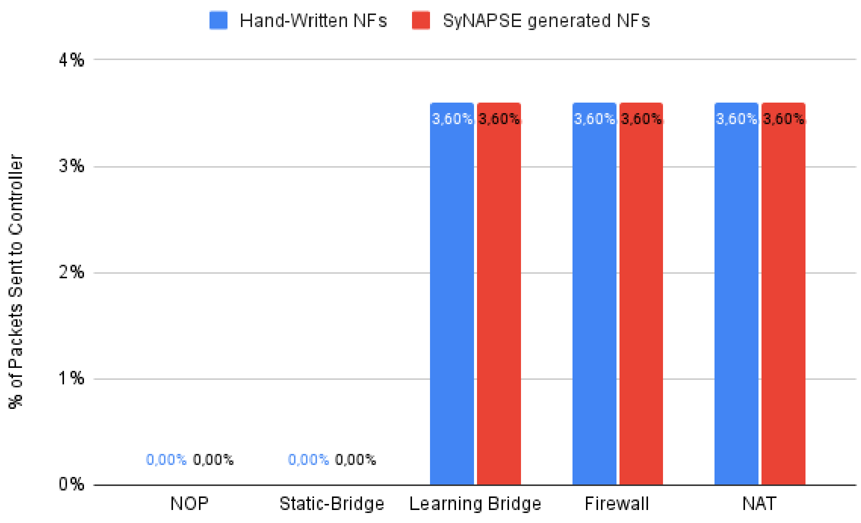


Figure 4.3: Comparison of the percentage of packets sent to the controller for all NFs, with a 45 second expiration time and 1 million packet with Uniform distribution.

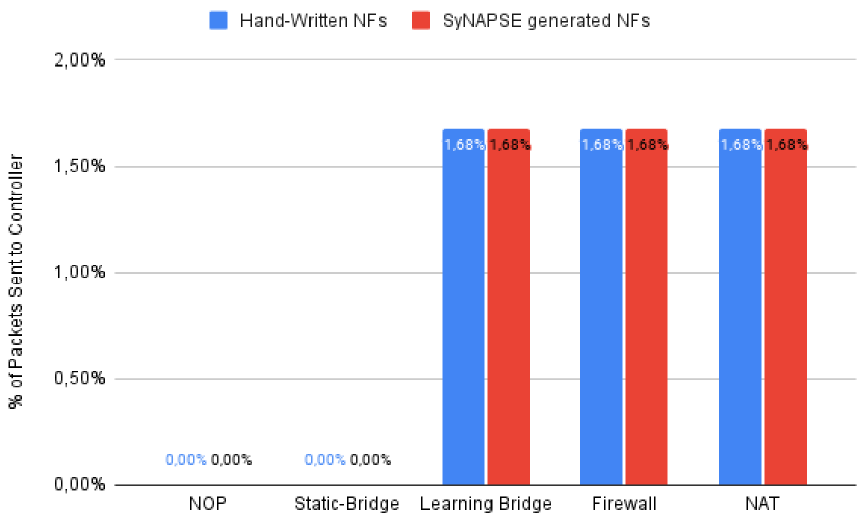


Figure 4.4: Comparison of the percentage of packets sent to the controller for all NFs, with a 15 second expiration time and 1 million packet with Zipfian distribution.

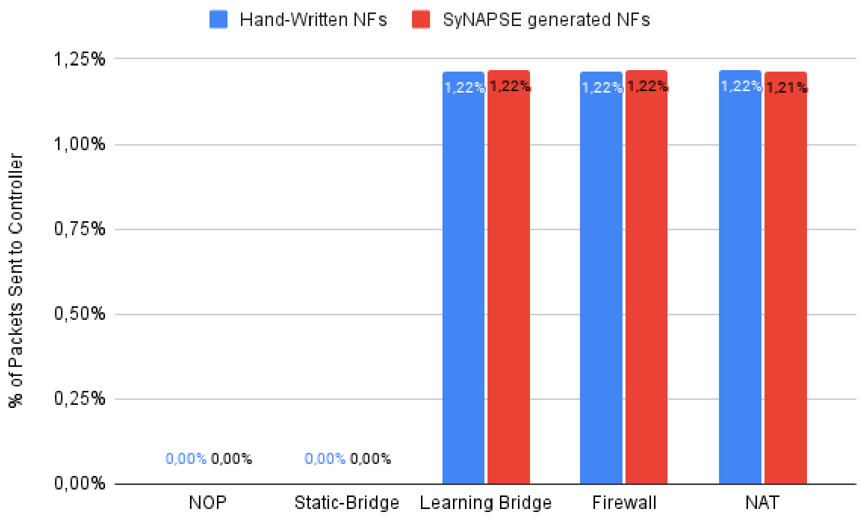


Figure 4.5: Comparison of the percentage of packets sent to the controller for all NFs, with a 45 second expiration time and 1 million packet with Zipfian distribution.

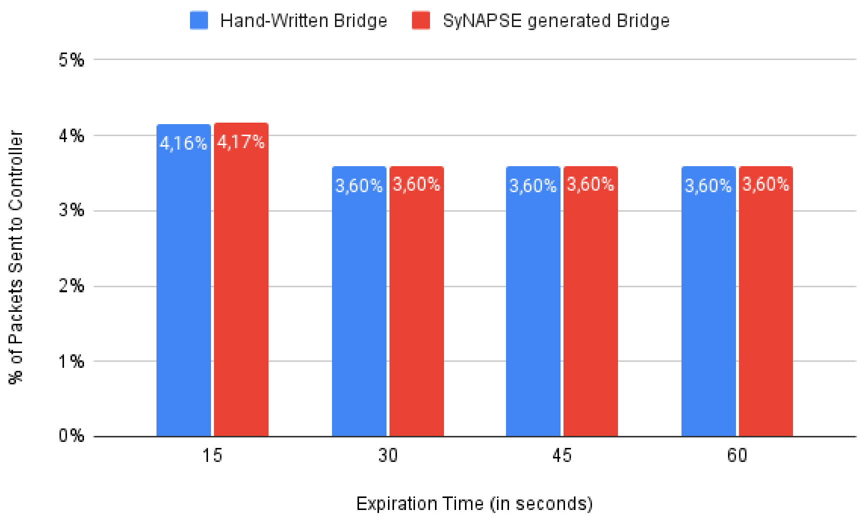


Figure 4.6: Percentage of packets sent to the controller for different expiration times in the Learning Bridge NF (1 Million packets with a Uniform Distribution).

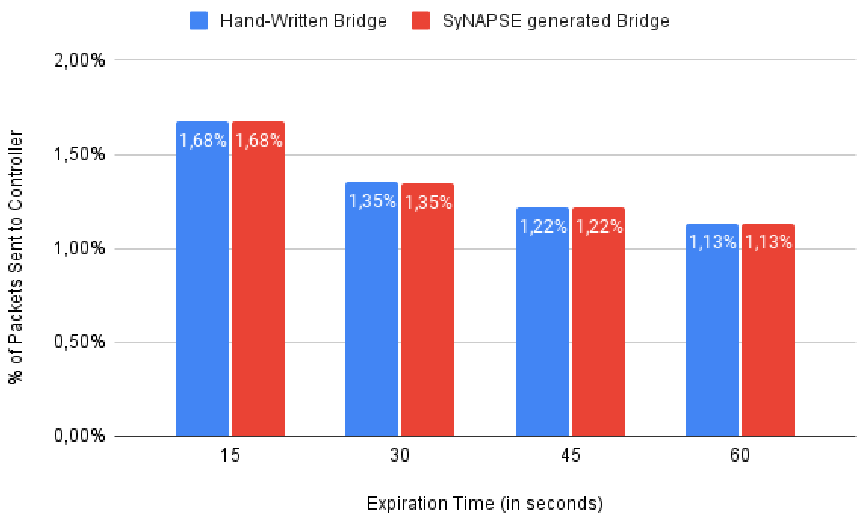


Figure 4.7: Percentage of packets sent to the controller for different expiration times in the Learning Bridge NF (1 Million packets with a Zipfian Distribution).

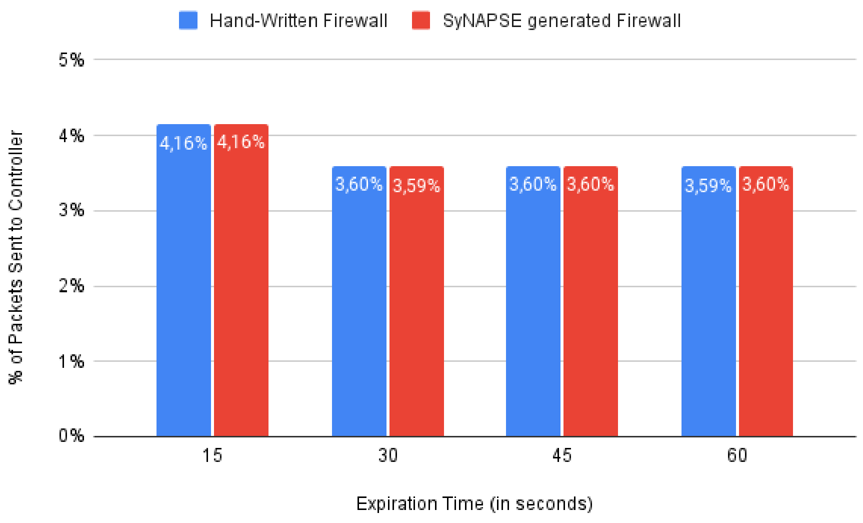


Figure 4.8: Percentage of packets sent to the controller for different expiration times in the Firewall NF (1 Million packets with a Uniform Distribution).

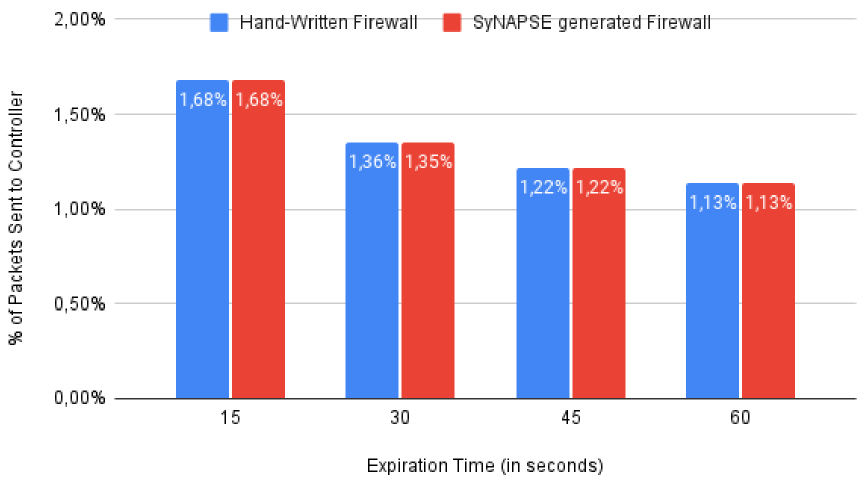


Figure 4.9: Percentage of packets sent to the controller for different expiration times in the Firewall NF (1 Million packets with a Zipfian Distribution).

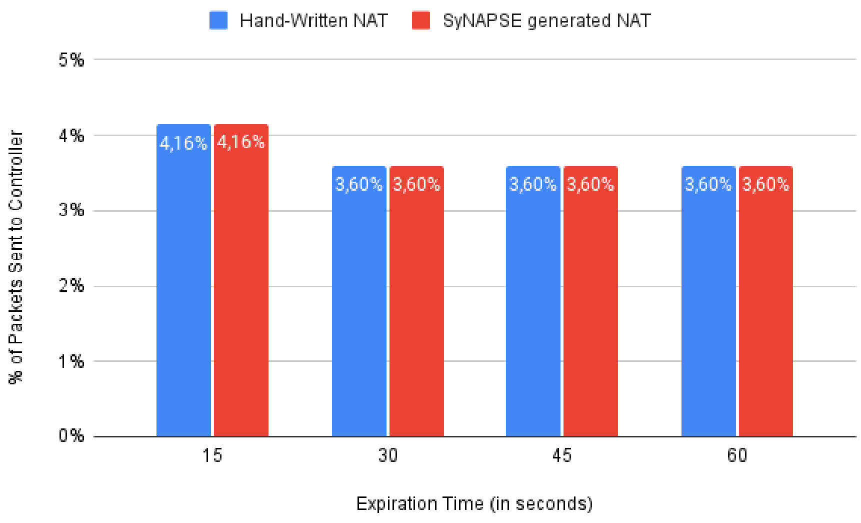


Figure 4.10: Percentage of packets sent to the controller for different expiration times in the NAT NF (1 Million packets with a Uniform Distribution).

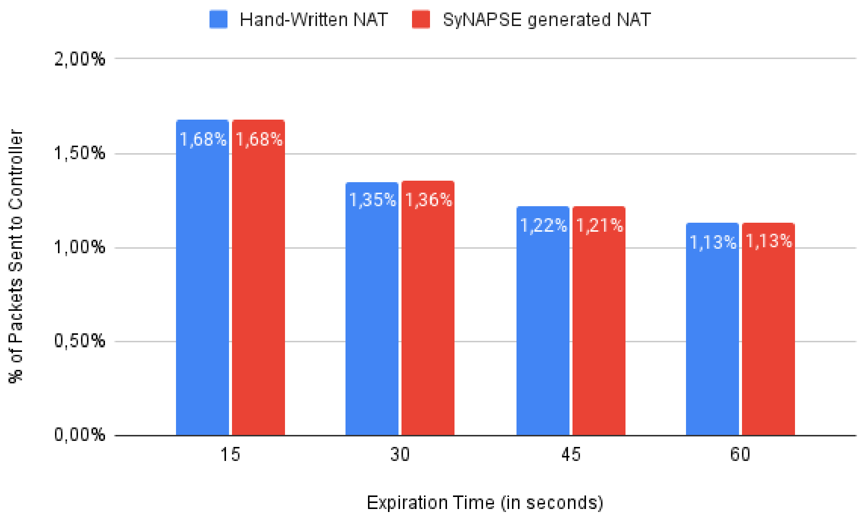


Figure 4.11: Percentage of packets sent to the controller for different expiration times in the NAT NF (1 Million packets with a Zipfian Distribution).

and written during packet forwarding and used as keys in the match-action tables.

In this section, we are going to carry out the following exercise: use the hand-written advanced NF implementations to **infer** the performance of the optimized implementations that SyNAPSE would generate. To derive these performance indicators we evaluated the hand-written advanced NFs and applied the same principles as in 4.3, where we analysed the controller interaction and applied a weighted average to deduce the NF's throughput.

Let us analyse how P4 registers were used in these implementations. To emulate the behaviour of a P4 table we used two P4 registers. For each flow, we use P4s hash mechanism to hash the 5-tuple and retrieve an index. By storing information in this index position we are able to check if a flow has been previously parsed by the NF. We use the first P4 register to store the timestamp of the last ”hit” for each flow entry and the second P4 register to store the flow addresses and ports. Whenever a new packet arrives to the switch we hash its 5-tuple to find its corresponding index. Depending on the NF functionality and in the content stored in the register we can retrieve/alter the data relative to the flow and apply packet forwarding actions. To apply these actions a set of requirements are necessary, for example we can forward the packet, if the flow is not expired and data stored in the second register contains data checks out against the packet that's being processed. Whenever different flows result in the same index (”collision”), we send the packet to the controller.

In the following graphs we compare all NF versions: the implementations generated by SyNAPSE, the previously studied Hand-written (Simple) implementations and their respective optimization (advanced). We can affirm that using this approach, where the registers act as a cache memory, we are able to significantly the number of controller interactions.

Throughput Estimation. We now infer the throughput of the NF advanced versions Regarding the estimate throughput for the stateless NFs like the NOP and the Static-Bridge, where no packet is sent to the controller, we obtained the same result of 3.2 Tbps.

$$3200 * (1 - 0.00) + 1 * 0.00 = 3200\ Gbps \tag{4.5}$$

In the Learning Bridge, we observe that only 0,04% of the packets are sent to the controller, resulting in an average throughput of 3187,204 Gbps.

$$3200 * (1 - 0.004) + 1 * 0.004 = 3187,204\ Gbps \tag{4.6}$$

The NAT and the Firewall implementations also reported that 1,68% of the packets are sent to the controller, thus the resulting throughput is 99,604 Gbps.

$$100 * (1 - 0.004) + 1 * 0.004 = 99,604\ Gbps \tag{4.7}$$

NF Type	Optimized NFs Throughput (Gbps)	SyNAPSE NFs Throughput (Gbps)	Maestro NFs Throughput (Mbps)
NOP	3200.0	3200.0	15
Static-Bridge	3200.0	3200.0	14
Learning-Bridge	3187.2	3146.62	-
Firewall	99.6	98.34	13
NAT	99.6	98.34	12.9

Table 4.2: Comparing throughput estimation for simple and advanced SyNAPSE NFs and the Maestro NFs.

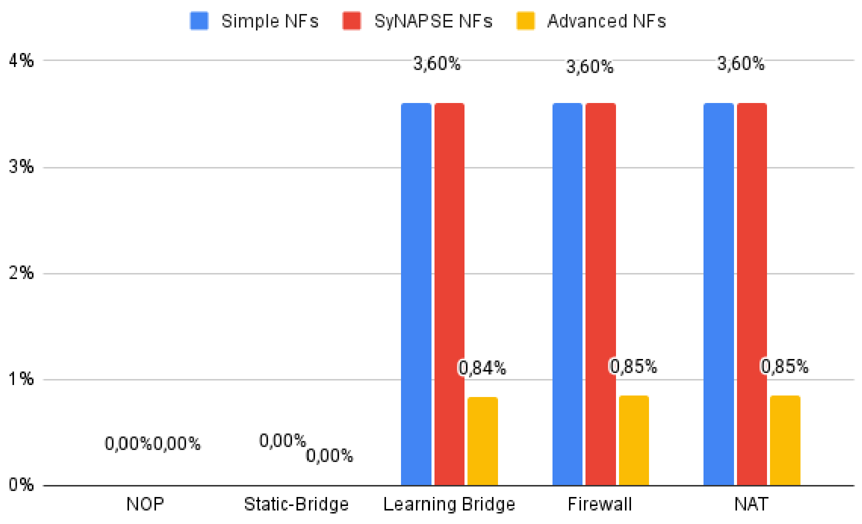


Figure 4.12: 45 second expiration time and Uniform distribution.

We can compare all NF versions throughput estimation in 4.2. These estimations results we have calculated improve the values obtained in the previous section and show us the potential for SyNAPSE, if it were able to automatically synthesize optimized NF implementations. We leave these tasks for Future Work 5.2.

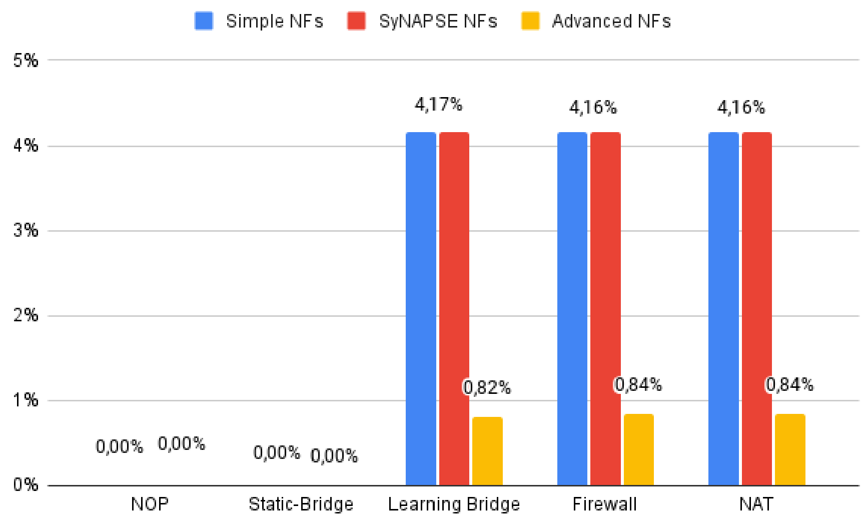


Figure 4.13: 15 second expiration time and Uniform distribution.

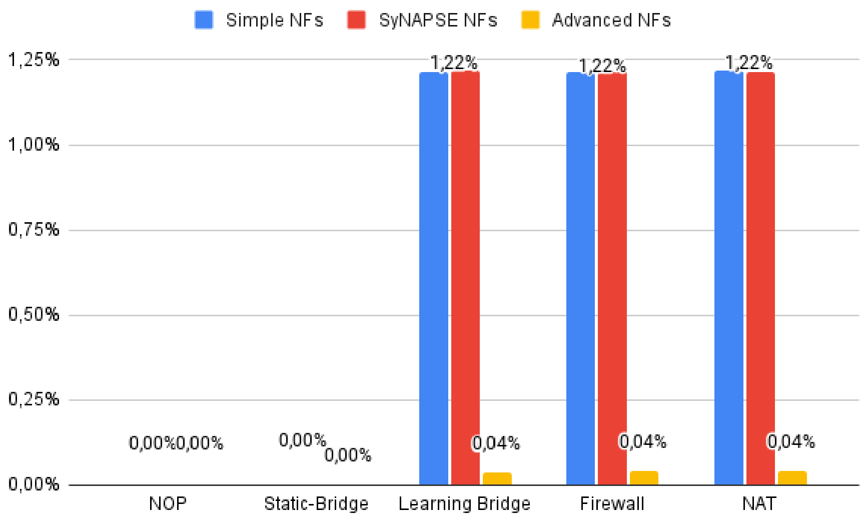


Figure 4.14: 45 second expiration time and Zipfian distribution.

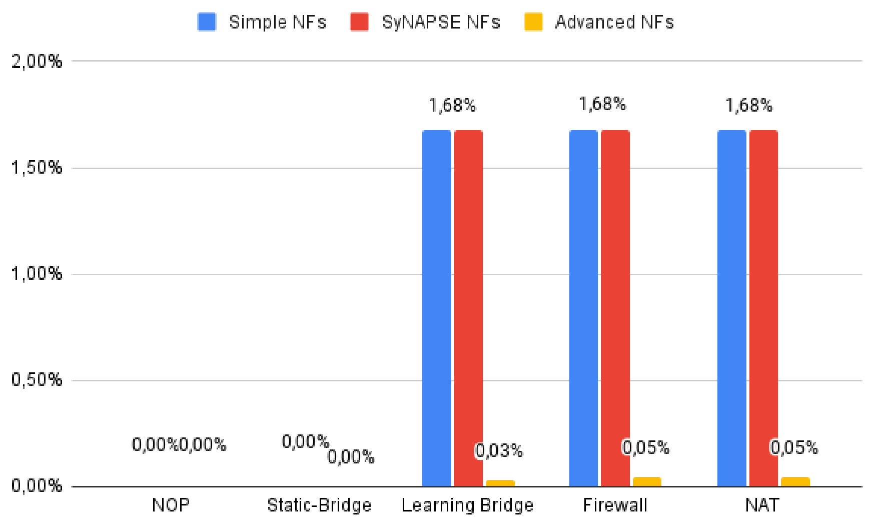


Figure 4.15: 15 second expiration time and Zipfian distribution.

5 Conclusion

Contents

5.1 Conclusions

This dissertation contributes to the state of the art of program synthesis and Software-Defined Networks (SDN) by introducing SyNAPSE, a tool capable of generating NF implementations that are split between a software P4 switch and a x86 controller, with the goal of offloading most of the packet-processing to the switch. These new NFs are synthesized from NFs programmed in the C language in the context of the Vigor framework, allowing non-network developers to benefit from the upsides of P4 without actually having to learn how to program in a network-oriented, low-level language.

To synthesize semantically equivalent NF implementations from NFs programmed in C, the SyNAPSE prototype gradually processes the NF code resorting to three different modules: NF Analysis module, Search Engine and Code Generator. In the NF analysis module we resort to Vigor's KLEE symbolic engine to obtain a complete and sound representation of the NFs behaviour. With this representation we make use of Maestro's to generate a functional model - a Binary decision diagram (BDD) - that represents all abstract functionality of the NF.

The BDD is introduced to the Search engine, the next component of the pipeline, where we'll apply synthesis-based techniques to generate an Execution Plan (EP). This intermediate representation can describe concretely the set of operations necessary to be performed by the x86 controller and P4 target platform. The Search Engine will produce multiple EP solutions, from which its heuristic mechanism will prioritize the solutions that are more reliant on the P4 switch target. To finally generate the C controller program and the P4 switch program, we pass the Execution Plan to the Code Generator. The first task of this component is to partition the Execution Plan according to the target platforms of each node. The information gathered from the EP nodes is categorized and subsequently used to complete program templates of a P4 program and a SDN controller program. These templates consist in lines of code that are common to all NF implementations. The final result will be two programs that represent a semantically equivalent NF of the input Vigor NFs

We evaluated our solution by comparing 5 different NFs implemented using SyNAPSE with hand-written implementations. The results show that SyNAPSE is able to synthesize NF implementations that, just like the hand-written ones, only interact with the x86 controller when a new flow reaches the software switch. In other words, we move most processing to the switch. Thus, we can conclude that SyNAPSE allows developers to make use of P4 software switches with no previous expertise.

5.2 Limitations and Future Work

Due to time constraints and issues with the tools we used, our implementation and evaluation still have some limitations. First, due to issues with the Barefoot proprietary software (the Software Development Kit), the generation of P4 code targeting a real hardware switch, following the the Tofino Native Architecture (TNA), was abandoned. We developed simple hand-written NFs targeting the TNA and Modules that targeted the TNA architecture by adapting the existing P4 Software Switch Modules, but we have not been able to move much forward on integrating it into the SyNAPSE prototype.Additionally, we would also need to extend the compatibility of the SyNAPSE Runtime API to the Tofino Switch.

In our initial evaluation our goal was to analyse the systems performance with real network traces. In the process we faced limitations in the latest stable versions of the BMv2 Simple Switch GRPC implementations, that inhibited us from performing the experiments we originally planned for (consisting of around 12 million packets each). We have anyway tweaked some configuration parameters the BMv2 swich and we were able to inject about 1 million packets per experiment.

In the section 4.4, we concluded that the use of P4 externs to minimize controller interaction produces favourable results. The strategy we have implemented to reduce the controller interaction includes the combined use of P4 tables and P4 registers, with slight adjustments to the type of information we store in the P4 registers, depending on the NF functionality. We leave this challenge of generating optimized NF implementation and finding a generic solution that can be applied to any libVig function (that currently generates a P4 table) for future work.

Bibliography

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[3] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[5] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," 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*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–295. [Online]. Available: https://doi.org/10.1145/3387514.3405869

[6] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 275–290. [Online]. Available: https://doi.org/10.1145/3341301.3359647

[7] K. Subramanian, L. D'Antoni, and A. Akella, "Genesis: Synthesizing forwarding tables in multi-tenant networks," *SIGPLAN Not.*, vol. 52, no. 1, p. 572–585, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3093333.3009845

[8] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," Sep 2020. [Online]. Available: https://www.techrxiv.org/articles/preprint/The_Programmable_Data_Plane_Abstractions_Architectures_Algorithms_and_Applications/12894677/1

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[10] M. Budiu and C. Dodd, "The p416 programming language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 5–14, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3139645.3139648

[11] G. ETSI, "002,"network functions virtualisation (nfv); architectural framework."," *Group Specification*, 2014.

[12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13–24. [Online]. Available: https://doi.org/10.1145/2342356.2342359

[13] H. Bi and Z.-H. Wang, "Dpdk-based improvement of packet forwarding," *ITM Web of Conferences*, vol. 7, p. 01009, 01 2016.

[14] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[15] D. Stancevic, "Zero copy i: user-mode perspective," *Linux Journal*, vol. 2003, no. 105, p. 3, 2003.

[16] F. Pereira, "Parallel NF synthesis," Master's Thesis, Instituto Superior Técnico / University of Lisbon, Jan. 2021.

[17] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: https://doi.org/10.1561/2500000010

[18] S. Gulwani and P. Jain, "Programming by examples: Pl meets ml," 11 2017, pp. 3–20.

[19] A. Solar-Lezama, “Program sketching,” *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 475–495, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0249-7

[20] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, “Network configuration synthesis with abstract topologies,” *SIGPLAN Not.*, vol. 52, no. 6, p. 437–451, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062367

[21] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, “Don’t mind the gap: Bridging network-wide objectives and device-level configurations,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 328–341. [Online]. Available: https://doi.org/10.1145/2934872.2934909

[22] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, “Network-wide configuration synthesis,” in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 261–281.

[23] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, “Netcomplete: Practical network-wide configuration synthesis with autocompletion,” in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’18. USA: USENIX Association, 2018, p. 579–594.

[24] S. Saha, S. Prabhu, and P. Madhusudan, “Netgen: Synthesizing data-plane configurations for network policies,” in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2774993.2775006

[25] Y. Yuan, D. Lin, R. Alur, and B. T. Loo, “Scenario-based programming for sdn policies,” in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2716281.2836119

[26] [Online]. Available: https://github.com/noxrepo/pox

[27] J. McClurg, H. Hojjat, and P. Černý, “Synchronization synthesis for network programs,” in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 301–321.

[28] Y. Wang, C. Jiang, X. Qiu, and S. G. Rao, “Learning network design objectives using a program synthesis approach,” in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 69–76. [Online]. Available: https://doi.org/10.1145/3365609.3365861

[29] H. Chen, A. Wang, and B. T. Loo, “Towards example-guided network synthesis,” in *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, ser. APNet ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 65–71. [Online]. Available: https://doi.org/10.1145/3232565.3234462

[30] X. Gao, T. Kim, A. K. Varma, A. Sivaraman, and S. Narayana, “Autogenerating fast packet-processing code using program synthesis,” in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 150–160. [Online]. Available: https://doi.org/10.1145/3365609.3365858

[31] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, “Switch code generation using program synthesis,” 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*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 44–61. [Online]. Available: https://doi.org/10.1145/3387514.3405852

[32] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, “Packet transactions: High-level programming for line-rate switches,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 15–28. [Online]. Available: https://doi.org/10.1145/2934872.2934900

[33] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, “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*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 435–450. [Online]. Available: https://doi.org/10.1145/3387514.3405879

[34] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, “Automated synthesis of adversarial workloads for network functions,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 372–385. [Online]. Available: https://doi.org/10.1145/3230543.3230573

[35] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’10.

New York, NY, USA: Association for Computing Machinery, 2010, p. 267–280. [Online]. Available: https://doi.org/10.1145/1879141.1879175

[36] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Passive and Active Measurement*, N. Taft and F. Ricciato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–95.